TOTUL DESPRE

C și C++

Manualul fundamental de programare în C și C++

Dr. Kris Jamsa
& Lars Klander

Pe CD-ROM:
Borland
Turbo C++ Lite –
Tot ce vă trebuie
pentru a crea
programe în C++

Teora

JAMSA

internet: www.teora.ro

TOTUL DESPRE

C și C++

Manualul fundamental de programare în C și C++

Dr. Kris Jamsa
& Lars Klander

Traducere de Eugen Dumitrescu

Teora

Titlul original: Jamsa's C/C++ Programmer's Bible

Teora
Calea Moşilor nr. 211, sector 2, Bucureşti
Fax: 01/210.38.28
e-mail: teora@teora.kappa.ro

Teora – Cartea prin poştă
CP 79 – 30, cod 72450 Bucureşti, România
Tel: 01/252.14. 31
e-mail: cpp@teora.kappa.ro

Copertă: Gheorghe Popescu
Tehnoredactare: Techno Media
Director Editorial: Diana Rotaru

Preşedinte: Teodor RĂDUCANU

NOT 4576 CAL C/C++, TOTUL DESPRE
ISBN: 973-601-911-X

Printed in Romania

Cuprins

Primele noțiuni de C

Macroinstrucțiuni și constante

Șiruri

Funcţii

Operații de la tastatură

Funcții matematice

Fișiere, directoare și discuri

Matrice, pointeri și structuri

Servicii DOS și BIOS

Gestionarea memoriei

Data și ora

Redirectarea intrărilor și ieșirilor și procesarea liniei de comandă

Instrumente de programare

Limbajul C avansat

Introducere în C++

Obiecte

Funcții de clasă uzuale

I/O în C++

Moștenire și polimorfism

Funcții generice și șabloane

Tratarea excepțiilor și portabilitatea tipurilor

Crearea unui exemplu de clasă reutilizabilă

Biblioteca standard de șabloane

Introducere în programarea în Windows

Mesaje și meniuri

Casete de dialog

Gestionarea memoriei în Windows

Procese și fire de execuție

Interfața cu dispozitivele grafice

Blocuri de bitmap, metafișiere și pictograme

Intrări/ieșiri în Windows

INTRODUCERE ÎN PROGRAMARE

C/C++ 1

Programele de calculator, cunoscute și sub numele de *software*, sunt constituite dintr-o serie de instrucțiuni pe care le execută calculatorul. Când creați un program, trebuie să specificați instrucțiunile pe care calculatorul trebuie să le execute pentru a realiza operațiile dorite. Procesul de definire a instrucțiunilor pe care le execută calculatorul este numit *programare*. Când creați un program, instrucțiunile se păstrează într-un fișier ASCII al cărui nume conține, de obicei, extensia C pentru un program C și extensia CPP pentru un program C++. De exemplu, dacă ați creat un program care realizează statul de plată, puteți să denumiți fișierul care conține instrucțiunile *stat.c*. C și C++ sunt doar două din multitudinea de limbaje de programare. Mulți programatori folosesc limbaje de programare ca BASIC, PASCAL și FORTRAN. Fiecare limbaj de programare are facilități specifice și propriile puncte forte (dar și slăbiciuni). Oricum, limbajele de programare există pentru a ne permite să definim instrucțiunile pe care vrem să le execute calculatorul.

Instrucțiunile pe care le execută un calculator sunt de fapt grupuri de 1 și 0 (cifre binare) care reprezintă semnale electronice produse în interiorul calculatorului. Pentru a programa primele calculatoare (în anii 1940-1950), programatorii trebuiau să înțeleagă modul în care calculatorul interpreta diferitele combinații de 0 și 1, deoarece programatorii scriau toate programele folosind cifre binare. Cum programele deveneau din ce în ce mai mari, acest mod de lucru a devenit foarte incomod pentru programatori. De aceea cercetătorii au creat limbaje de programare care permit exprimarea instrucțiunilor calculatorului într-o formă mai accesibilă omului. După ce programatorii scriau instrucțiunile într-un fișier (numit *fișier sursă*), un al doilea program (numit *compilator*), convertea instrucțiunile limbajului de programare în șirurile de 1 și 0 (cunoscute sub numele de *cod mașină*), pe care le putea înțelege calculatorul. Fișierele dumneavoastră cu extensia EXE sau COM conțin codul mașină (șiruri de 1 și 0) pe care calculatorul îl va executa. Figura 1 ilustrează procesul de compilare a unui fișier sursă, în urma căruia se obține un program executabil.

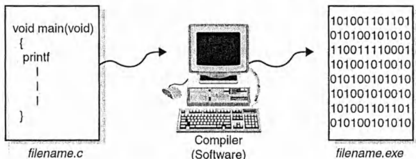


***Figura 1**: Un compilator convertește instrucțiunile din cod sursă în cod mașină.*

După ce ați creat un fișier sursă, rulați un compilator pentru a converti instrucțiunile într-un format pe care calculatorul poate să-l execute. Dacă, de exemplu, utilizați produsul *Turbo C++ Lite*™ al firmei Borland (inclus pe discul CD-ROM care însoțește această carte), veți apela compilatorul cu opțiunea *Compile to OBJ* din meniul Compile (ceea ce înseamnă a da instrucțiunea de compilare a fișierului sursă). Următoarele capitole vă indică pașii pe care trebuie să-i executați pentru a crea și a compila un program în C.

2 CREAREA UNUI FIȘIER SURSĂ ASCII



Când creați un program, instrucțiunile pe care doriți să le execute calculatorul trebuie plasate într-un fișier numit *fișier sursă*. Dacă nu folosiți *Turbo C++ Lite* sau un compilator-editor performant, trebuie să vă creați fișierele programului folosind un editor ASCII, cum ar fi programul EDIT pe care îl oferă sistemul de operare DOS. Nu puteți crea programe folosind un procesor de text (cum ar fi *Microsoft Word®* sau *Corel WordPerfect®*). După cum știți, procesoarele de text vă permit să formatați documentele, aliniind marginile, scriind caractere cursive sau subliniind textul și așa mai departe. Pentru a realiza toate aceste operații, procesoarele de text introduc caractere speciale în interiorul documentelor. Cu toate că aceste caractere au sens pentru procesorul de text, ele vor deruta compilatorul care convertește fișierul sursă în cod mașină și această confuzie va cauza erori. Când vă creați fișierul sursă, asigurați-vă că i-ați dat un nume care descrie cu acuratețe funcția programului. De exemplu, ar trebui să denumiți fișierul sursă pentru un program de gestiune *gestiune.c*, iar fișierul sursă pentru un joc *fotbal.c*.

Dacă, pe de altă parte, folosiți un compilator care include un editor intern, vă puteți crea programele în cadrul acestui editor. De exemplu, dacă folosiți *Turbo C++ Lite*, veți crea un nou fișier program cu opțiunea New din meniul File. Pentru a crea primul dumneavoastră program cu ajutorul produsului *Turbo C++ Lite*, trebuie să executați următorii pași:

1. Selectați meniul File, opțiunea New. *Turbo C++ Lite* va crea fișierul *noname00.cpp*.
2. În fereastra *noname00.cpp*, introduceți codul următor:

```
#include <stdio.h>
void main(void)
{
   printf("Totul despre C/C++");
}
```

3. Selectați meniul File, opțiunea Save As. *Turbo C++ Lite* va afișa caseta de dialog Save File As.
4. În caseta de dialog Save File As, introduceți numele *primul.c* și apăsați ENTER. *Turbo C++ Lite* va salva fișierul program *primul.c*.

Cu toate că programul *primul.c* conține șase linii, numai instrucțiunea *printf* are de fapt un rol activ. Cînd executați acest program, *printf* va afișa pe ecran mesajul *Totul despre C/C++*. Fiecare limbaj de programare (la fel ca și limba engleză, franceză sau germană) are un set de reguli, denumite *reguli sintactice*, căruia trebuie să vă conformați când folosiți respectivul limbaj. De asemenea, trebuie să vă supuneți regulilor sintactice ale limbajelor de programare C atunci când creați programe în C. Printre aceste reguli sintactice se numără introducerea parantezelor după numele *main* și utilizarea semnului punct și virgulă la sfârșitul instrucțiunii *printf*. Când scrieți programul, trebuie să aveți mare grijă să nu omiteți vreunul din aceste elemente. Verificați a doua oară ceea ce ați scris, pentru a vă asigura că ați introdus corect instrucțiunile programului C, exact așa cum apar în exemplul de mai sus. Dacă nu sunt greșeli, salvați conținutul fișierului pe disc. În următorul capitol, veți învăța cum să compilați fișierul sursă și să convertiți instrucțiunile programului C în limbajul mașină pe care calculatorul poate să-l înțeleagă și să-l execute.

COMPILAREA PROGRAMULUI ÎN C



În paragraful precedent, ați creat un fișier sursă în C, *primul.c*. Acesta conține instrucțiunea *printf*, care va afișa pe ecran mesajul *Totul despre C/C++* atunci când veți executa programul. Un fișier sursă conține instrucțiuni într-o formă pe care o puteți înțelege (sau cel puțin o veți putea înțelege după ce veți învăța limbajul C). Un program executabil, pe de altă parte, conține instrucțiuni exprimate ca șiruri de 1 și 0, pe care le înțelege calculatorul. Procesul de conversie a fișierului sursă C în limbaj mașină este cunoscut sub numele de *compilare*. În funcție de compilatorul C pe care-l folosiți, comanda care trebuie executată pentru a compila fișierul sursă va fi diferită. Presupunând că utilizați produsul *Turbo C++ Lite* al firmei Borland, puteți compila programul pe care l-ați creat în secțiunea 2 (*primul.c*) folosind următoarea secvență de comenzi:

1. Selectați meniul Compile, opțiunea Build All. *Turbo C++ Lite* va afișa caseta de dialog Compiling (compilare).
2. Dacă operația de compilare se încheie cu succes, compilatorul va afișa mesajul *Press any key* (apăsați orice tastă). În cazul în care compilatorul nu creează fișierul *primul.exe*, ci afișează mesaje de eroare pe ecran, probabil că ați încălcat o regulă sinctactică a limbajului, așa cum se explică în capitolul următor.
3. Dacă ați reușit să scrieți instrucțiunile în C exact ca în secțiunea 2, compilatorul va crea un fișier executabil, denumit *primul.exe*. Pentru a executa programul *primul.exe*, puteți să selectați opțiunea Run din meniul Run sau să folosiți combinația de taste Ctrl + F9.

Când executați programul, ecranul va afișa următorul rezultat:

```
Totul despre C/C++
C:\>
```

Observație: *În anumite implementări, **Turbo C++ Lite** va genera rezultatul și se va întoarce imediat la fereastra de editare. În aceste cazuri, selectați meniul File, opțiunea DOS Shell pentru a vedea rezultatul programului.*

ERORILE DE SINTAXĂ



Așa cum ați citit în secțiunea 2, fiecare limbaj de programare are un set de reguli, denumite *reguli sintactice*, pe care trebuie să îl respectați atunci când introduceți instrucțiunile programului dumneavoastră. Dacă încălcați o regulă sintactică, programul nu va fi compilat cu succes. Într-o astfel de situație, compilatorul va afișa pe ecran mesaje de eroare ce precizează linia din programul dumneavoastră care conține eroarea, precum și o scurtă descriere a greșelii. Folosiți editorul pentru a crea fișierul *sintaxa.c*, care conține o eroare de sintaxă. În următorul exemplu, programului îi lipsesc ghilimelele de închidere a mesajului *Totul despre C/C++*:

```
#include <stdio.h>
void main(void)
  {
     printf("Totul despre C/C++);
  }
```

Când compilați acest program, compilatorul va afișa un mesaj de eroare sintactică atunci când întâlnește linia 5. În funcție de compilatorul dumneavoastră, mesajul de eroare va fi diferit. Dacă utilizați *Turbo C++ Lite*, ecranul dumneavoastră va afișa următoarele mesaje de eroare:

```
Error sintaxa.c 5: Unterminated string or character constant in
    function main
Error sintaxa.c 6: Function call missing ) in function main()
Error sintaxa.c 6: Statement missing ; in function main()
```

Cu toate că în codul sursă *sintaxa.c* există doar o eroare, compilatorul de C va afișa trei mesaje de eroare. Lipsa ghilimelelor de închidere provoacă o serie de erori în cascadă în cadrul compilării (o eroare duce la alta).

Pentru a corecta erorile de sintaxă din programele dumneavoastră, urmați pașii indicați mai jos:

1. Notați-vă numărul liniei fiecărei erori și adăugați o scurtă descriere.
2. Editați fișierul sursă mutând cursorul la prima linie indicată de compilator.
3. În fișierul sursă, corectați eroarea și mutați cursorul la următoarea linie. Cele mai multe editoare afișează numărul de ordine al liniei curente, pentru a vă ajuta să localizați liniile din fișier.

În cazul fișierului *sintaxa.c*, editați fișierul și adăugați ghilimelele lipsă. Salvați fișierul pe disc și folosiți compilatorul pentru a-l compila. După ce ați corectat eroarea, compilatorul va crea fișierul *sintaxa.exe*. Pentru a executa *sintaxa.exe*, selectați meniul Run, opțiunea Run. Programul va rula și va produce rezultatul de mai jos:

```
Totul despre C/C++
C:\>
```

5 STRUCTURA UNUI PROGRAM TIPIC ÎN C

C/C++

În secțiunea 2, ați creat fișierul sursă *primul.c*, care conținea următoarele instrucțiuni:

```
#include <stdio.h>
void main(void)
  {
    printf("Totul despre C/C++");
  }
```

Aceste instrucțiuni sunt similare cu cele pe care le veți întâlni în majoritatea programelor în C. În multe cazuri, un fișier sursă în C trebuie să înceapă cu una sau mai multe instrucțiuni *#include*. Instrucțiunea *#include* cere compilatorului de C să folosească conținutul unui anumit fișier. În cazul fișierului *primul.c*, instrucțiunea *#include* cere compilatorului să folosească un fișier denumit *stdio.h*. Fișierele pe care le specifică instrucțiunea *#include* sunt fișiere ASCII, care conțin cod sursă în C. Puteți tipări sau vizualiza conținutul oricărui fișier, urmând pașii indicați în capitolul 13. Fișierele pe care le apelați cu o directivă *#include* (de obicei au extensia *h*) sunt denumite *fișiere include* sau *fișiere antet*. Cele mai multe dintre fișierele antet conțin instrucțiuni pe care programele dumneavoastră le folosesc în mod uzual, dar veți învăța mai târziu în această carte și despre alte utilizări ale fișierelor antet. Când indicați compilatorului de C să includă conținutul unui fișier, nu mai trebuie să scrieți

dumneavoastră instrucțiunile respective în program. După directiva *#include*, de obicei veți întâlni o instrucțiune de genul:

```
void main(void)
```

Fiecare program în C pe care îl creați va avea o linie similară instrucțiunii *void main*. Așa cum ați citit în capitolul 1, un program în C conține o listă de instrucțiuni pe care doriți să le execute calculatorul. Pe măsură ce complexitatea programelor dumneavoastră va crește, le veți împărți în mici părți care sunt mai simplu de înțeles pentru dumneavoastră (și pentru ceilalți care vă citesc programul). Grupul de instrucțiuni pe care doriți să le execute calculatorul mai întâi se numește *programul principal*. Instrucțiunea *void main* identifică acest grup de instrucțiuni (programul principal) pentru compilator.

Bineînțeles, deoarece compilatorul de C va împărți instrucțiunile în două categorii, cele care alcătuiesc programul principal și cele care sunt suplimentare, va trebui să aveți o posibilitate de a înștiința compilatorul de C care instrucțiuni corespund fiecărei părți a programului. Pentru a asocia instrucțiunile programului unei anumite secțiuni, plasați instrucțiunile între două acolade ({}). Acoladele sunt părți ale sintaxei limbajului C. Pentru fiecare acoladă deschisă, trebuie să aveți perechea ei, care închide grupul de instrucțiuni.

COMPLETAREA INSTRUCȚIUNILOR PROGRAMULUI

C/C++ 6

După cum ați văzut, programul *primul.c* a folosit instrucțiunea *printf* pentru a afișa un mesaj pe ecran. Următorul program în C, *3_mesaje.c*, folosește trei instrucțiuni *printf* pentru a afișa același mesaj. Toate instrucțiunile se află între acoladele de deschidere și închidere ale programului:

```
#include <stdio.h>

void main(void)
   {
     printf("Totul");
     printf("despre");
     printf("C/C++");
   }
```

Observați caracterul spațiu din interiorul instrucțiunii *printf*. Acest spațiu este important, deoarece el ne asigură că programul va afișa corect textul pe ecran (plasând un spațiu între cuvinte). Pe măsură ce numărul de instrucțiuni din program crește, crește și probabilitatea erorilor sintactice. Verificați încă o dată programul dumneavoastră pentru a vă asigura că ați scris corect fiecare instrucțiune, apoi salvați fișierul pe disc. Când compilați și executați programul *3_mesaje*, ecranul dumneavoastră va afișa următorul rezultat:

```
Totul despre C/C++
C:\>
```

AFIȘAREA IEȘIRII PE O LINIE NOUĂ

C/C++ 7

Cele mai multe dintre programele anterioare au afișat mesajul *Totul despre C/C++* pe ecranul monitorului dumneavoastră. Pe măsură ce programele vor deveni mai complexe, probabil

că veți dori ca ele să afișeze rezultatele pe două sau mai multe linii. În capitolul 6, ați creat programul *3_mesaje.c*, care a folosit trei instrucțiuni *printf* pentru a afișa un mesaj pe ecran:

```
printf("Totul ");
printf("despre ");
printf("C/C++");
```

Dacă nu îi cereți să facă altfel, *printf* va continua să afișeze rezultatul pe linia curentă. Scopul următorului program, *o_linie.c*, este să afișeze rezultatul pe două linii succesive.

```
#include <stdio.h>

void main(void)
  {
    printf("Aceasta este linia unu.");
    printf("Aceasta este linia a doua.");
  }
```

Când compilați și executați programul *o_linie.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Aceasta este linia unu. Aceasta este linia a doua.
C:\>
```

Când doriți ca *printf* să înceapă afișarea pe o linie nouă, trebuie să includeți un caracter special, *linie nouă* (\n), în interiorul textului pe care *printf* urmează să-l afișeze. Când *printf* întâlnește caracterul \n, va avansa cursorul la începutul liniei următoare. Programul de mai jos, *doua_lin.c*, folosește caracterul linie nouă pentru a afișa linia a doua de text pe o nouă linie, așa cum s-a dorit:

```
#include <stdio.h>

void main(void)
  {
    printf("Aceasta este linia unu.\n");
    printf("Aceasta este linia a doua.");
  }
```

Atunci când compilați și executați programul *doua_lin.c*, pe ecranul dumneavoastră se va afișa:

```
Aceasta este linia unu.
Aceasta este linia a doua.
C:\>
```

Multe dintre programele acestei cărți folosesc caracterul linie nouă. Practic, fiecare program pe care îl veți scrie va folosi caracterul linie nouă în unul sau mai multe locuri.

8 C FACE DIFERENȚA ÎNTRE MINUSCULE ȘI MAJUSCULE



Atunci când vă scrieți programele, nu trebuie să uitați că C consideră literele mari și cele mici ca fiind diferite. De regulă, cele mai multe comenzi C folosesc minuscule, cele mai multe constante C sunt scrise în întregime cu majuscule, iar cele mai multe variabile folosesc un amestec de litere mari și mici. Programele în C folosesc mai ales literele mici. Deoarece

următorul program, *ermajusc.c*, folosește litera M în numele *Main* (în loc de *main*), compilarea nu se va încheia cu succes:

```
#include <stdio.h>

void Main(void)
  {
    printf("Acest program nu va fi compilat.");
  }
```

Când veți compila programul *ermajusc.c*, compilatorul *Turbo C++ Lite* va afișa următorul mesaj:

```
Linker error: Undefined symbol _main in module TURBO_C\C0S.ASM
```

Mesajul relativ lipsit de sens pe care compilatorul *Turbo C++ Lite* îl afișează este rezultatul scrierii cuvântului *Main* cu majuscula M. În acest caz, pentru a corecta eroarea, trebuie pur și simplu să înlocuiți *Main* cu *main*. După ce ați efectuat înlocuirea, recompilați și executați programul.

ERORILE LOGICE (BUGS)



În secțiunea 4, ați învățat că dacă încălcați una dintre regulile limbajului C, compilatorul va afișa un mesaj de eroare de sintaxă, iar programul nu se va compila. Pe măsură ce programele dumneavoastră vor deveni mai complexe, vor apărea situații în care programul se compilează cu succes, dar nu execută corect funcția pe care i-ați dat-o. De exemplu, să presupunem că doriți ca următorul program, *o_linie.c*, să afișeze rezultatul pe două linii.

```
#include <stdio.h>

void main(void)
  {
    printf("Aceasta este linia unu.");
    printf("Aceasta este linia a doua.");
  }
```

Deoarece acest program nu încalcă nici o regulă de sintaxă a limbajului C, programul se va compila cu succes. Totuși, când veți executa programul, el nu va afișa rezultatul pe două linii, ci îl va afișa pe o singură linie, așa cum se vede mai jos:

```
Aceasta este linia unu. Aceasta este linia a doua.
C:\>
```

Când programul dumneavoastră nu lucrează așa cum ați fi dorit, înseamnă că el conține *erori logice*, numite și *bugs*. Când programele dumneavoastră conțin erori logice (ele vor apărea în mod sigur), trebuie să încercați să descoperiți și să corectați cauza erorii. Procesul de înlăturare a erorilor logice dintr-un program este numit *depanare*. Veți învăța mai târziu în această carte diferite tehnici pe care le puteți folosi pentru a localiza erorile logice din programul dumneavoastră. Pentru început, totuși, cea mai bună cale de a localiza astfel de erori este tipărirea unei copii a programului și examinarea lui linie cu linie, până când veți localiza eroarea. Examinarea programului linie cu linie se numește *desk cheking*. În cazul programului *o_linie.c*, această verificare ar trebui să vă arate că prima instrucțiune *printf* nu conține caracterul linie nouă (\n).

10 DEZVOLTAREA PROGRAMULUI

C/C++

Atunci când creați programe, executați de obicei aceiași pași. La început, veți utiliza un editor pentru a crea fișierul sursă. După aceea, veți compila programul. Dacă programul conține erori sintactice, trebuie să editați fișierul sursă și să corectați erorile. După ce programul este compilat cu succes, veți încerca să rulați programul. Dacă programul se execută cu succes și realizează ceea ce era prevăzut, înseamnă că ați terminat de realizat programul. Pe de altă parte, dacă programul nu lucrează cum v-ați fi așteptat, trebuie să verificați linie cu linie codul sursă pentru a localiza eroarea logică (așa cum am văzut în secțiunea 9). După ce corectați eroarea, trebuie să compilați din nou codul sursă pentru a crea un nou fișier executabil. Apoi puteți testa noul program pentru a vă asigura că el realizează sarcina așa cum ați dorit. Figura 10 ilustrează procesul de dezvoltare a programului.

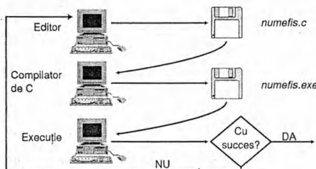


Figura 10 *Procesul de dezvoltare a programului*

11 TIPURILE DE FIȘIERE

C/C++

Atunci când creați un program în C, vă plasați instrucțiunile într-un fișier sursă care are extensia C. Dacă programul se compilează cu succes, compilatorul va crea un fișier executabil, cu extensia EXE. Așa cum ați citit în secțiunea 5, multe programe utilizează fișiere antet (cu extensia H), care conțin instrucțiuni utilizate în mod curent. Dacă examinați conținutul directorului dumneavoastră după ce ați compilat un program, veți găsi unul sau mai multe fișiere cu extensia OBJ. Aceste fișiere, denumite *fișiere obiect*, conțin instrucțiuni sub formă de șiruri de 0 și 1, pe care le înțelege calculatorul. Totuși, nu puteți executa aceste fișiere obiect, deoarece conținutul lor nu este chiar complet.

Compilatorul de C pune la dispoziție rutine (cum ar fi *printf*), care realizează operațiile frecvent utilizate și reduc numărul de instrucțiuni pe care trebuie să le introduceți în program. După ce compilatorul examinează sintaxa programului, el creează un fișier obiect. În cazul programului *primul.c*, compilatorul va crea un fișier obiect denumit *primul.obj*. După aceea, un program denumit *editor de legături* alcătuiește programul executabil prin combinarea instrucțiunilor programului din fișierul obiect cu funcțiile pe care le deține compilatorul (cum ar fi *printf*). De cele mai multe ori, atunci când apelați compilatorul pentru a vă examina fișierul sursă, acesta va apela în mod automat editorul de legături dacă programul s-a compilat cu succes. Figura 11 ilustrează procesul de compilare și de realizare a legăturilor unui program:

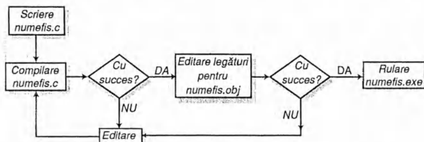


Figura 11 *Procesul de compilare a unui program și de editare a legăturilor.*

EDITORUL DE LEGĂTURI

C/C++ 12

În secțiunea 11, ați învățat că atunci când compilați un program în C, un al doilea program, denumit *editor de legături*, combină instrucțiunile programului dumneavoastră cu rutinele predefinite (pe care le deține compilatorul), pentru a converti fișierul obiect într-un program executabil. La fel ca în cazul procesului de compilare, în cursul căruia se pot detecta erori de sintaxă, procesul de editare a legăturilor poate, de asemenea, să întâlnească erori. Să luăm de exemplu următorul program, *nu_print.c*, care folosește, în mod eronat, *print* în loc de *printf*.

```
#include <stdio.h>
void main(void)
  {
    print("Pentru acest program, nu se executa editarea
    legaturilor");
  }
```

Pentru că programul *nu_print.c* nu încalcă nici o regulă de sintaxă a limbajului C, programul va fi compilat cu succes, rezultând un fișier de tip OBJ. Totuși, din cauza instrucțiunii *print* nedefinite, editorul de legături *Turbo C++ Lite* va afișa următorul mesaj de eroare:

```
Error: Function 'print' should have a prototype in function main()
```

Deoarece compilatorul de C nu deține nici o funcție denumită *print*, editorul de legături nu poate crea programul executabil *nu_print.exe*, însă va afișa mesajul de eroare de mai sus. Pentru a corecta eroarea, editați fișierul înlocuind *print* cu *printf* și recompilați programul.

FIȘIERELE ANTET

C/C++ 13

Fiecare program prezentat în cadrul acestei cărți folosește una sau mai multe instrucțiuni *#include* pentru a cere compilatorului de C să folosească instrucțiunile incluse într-un *fișier antet*. Un fișier antet este un fișier ASCII, al cărui conținut îl puteți imprima sau afișa pe ecranul dumneavoastră. Dacă examinați directorul care conține compilatorul dumneavoastră (directorul *tclite* în cazul compilatorului *Turbo C++ Lite* al firmei Borland), veți întâlni un subdirector denumit *include*. Subdirectorul *include* conține fișierele antet ale compila-

torului. Faceți-vă timp acum pentru a localiza fișierele antet ale compilatorului dumneavoastră. Poate că ar fi bine chiar să tipăriți conținutul unui fișier des utilizat, cum ar fi *stdio.h*. În interiorul acestui fișier, veți întâlni instrucțiuni în limbajul C. Când compilatorul întâlnește o instrucțiune *#include* în program, el compilează codul din fișierul antet ca și cum ați fi scris conținutul fișierul antet în fișierul sursă. Fișierele antet conțin definiții frecvent utilizate și furnizează compilatorului informații referitoare la funcțiile sale, cum ar fi *printf*. Deocamdată, probabil că veți înțelege mai greu conținutul unui fișier antet. Pe măsură ce vă veți experimenta în C și C++, vă recomandăm să tipăriți o copie a fiecărui fișier antet pe care îl folosiți și să-l examinați. Fișierele antet conțin informații prețioase și vă furnizează tehnici de programare care vă vor face să deveniți un programator mai bun.

14 LOCALIZAREA FIȘIERELOR ANTET



În secțiunea 13, ați învățat că atunci când compilatorul de C întâlnește o directivă *#include*, adaugă conținutul fișierului antet la programul dumneavoastră ca și cum l-ați fi scris în interiorul fișierului sursă. În funcție de compilatorul cu care lucrați, parametrii dumneavoastră de mediu trebuie să conțină o intrare INCLUDE care să indice compilatorului numele subdirectorului cu fișierele antet. Dacă, atunci când compilați un program, compilatorul afișează un mesaj de eroare, avertizând că nu poate deschide un anumit fișier antet, verificați mai întâi subdirectorul care conține fișierele antet ale compilatorului dumneavoastră, pentru a vă asigura că acel fișier există. Dacă întâlniți fișierul, dați comanda SET la promptul DOS, așa cum se vede mai jos:

```
C:\>SET <ENTER>
COMSPEC=C:\DOS\COMMAND.COM
PATH=C:\DOS;C:\WINDOWS;C:\BORLANDC\BIN
PROMPT=$P$G
TEMP=C:\TEMP
```

Dacă parametrii dumneavoastră de mediu nu conțin o intrare INCLUDE, verificați documentația care însoțește compilatorul dumneavoastră pentru a determina dacă este necesară o astfel de intrare. De obicei, în procesul de instalare a compilatorului, va fi introdusă în fișierul *autoexec.bat* o comandă SET care asociază variabilei INCLUDE subdirectorul care conține fișierele antet, ca mai jos:

```
SET INCLUDE=C:\BORLANDC\INCLUDE
```

În cazul în care compilatorul dumneavoastră folosește variabila INCLUDE și fișierul *autoexec.bat* nu definește această intrare, puteți să o creați dumneavoastră, plasând-o în fișierul *autoexec.bat*.

Observație: *Programul* ***Turbo C++ Lite*** *va căuta fișierele antet numai în interiorul subdirectorului său* ***include.***

15 MĂRIMEA VITEZEI DE COMPILARE



Atunci când compilați un fișier sursă, compilatorul de C creează unul sau mai multe fișiere temporare care există numai cât timp lucrează compilatorul și editorul de legături. În funcție de compilator, puteți utiliza variabila de mediu TEMP pentru a indica locul unde să fie plasate aceste fișiere temporare. Dacă folosiți un calculator cu mai multe hard-discuri, unele dintre ele având mai mult spațiu disponibil decât altele (în special atunci când compilatorul

dumneavoastră funcţionează în mediul Windows şi, prin urmare, foloseşte memorie virtuală şi fişiere de schimb), aţi putea atribui variabilei TEMP unitatea care are cel mai mare spaţiu liber. Astfel, compilatorul va crea fişierele sale temporare pe un hard-disc mult mai rapid, ceea ce va mări viteza procesului de compilare. Presupunând că unitatea D are acest spaţiu excedentar, o puteţi atribui variabilei TEMP plasând o comandă SET în fişierul dumneavoastră *autoexec.bat*, aşa cum se vede mai jos:

```
SET TEMP=D:
```

COMENTAREA PROGRAMELOR

C/C++ 16

Ca regulă, de fiecare dată când creaţi un program, trebuie să vă asiguraţi că aţi inclus în acesta *comentarii* care explică procesele pe care le execută programul. Pe scurt, un comentariu este un mesaj care vă ajută să citiţi şi să înţelegeţi programul. Pe măsură ce programele dumneavoastră vor creşte în lungime, vor deveni tot mai greu de înţeles. Deoarece veţi crea sute sau chiar mii de programe, nu veţi putea să vă amintiţi scopul fiecărei instrucţiuni din interiorul unui program. Dacă veţi include comentarii în program, nu va fi nevoie să vă reamintiţi fiecare detaliu al programului, pentru că aceste comentarii vă vor furniza explicaţiile necesare.

Cele mai multe dintre compilatoarele recente de C şi C++ oferă două modalităţi de a insera comentarii în interiorul fişierelor sursă. Prima modalitate este plasarea a două caractere slash (//), aşa cum se vede mai jos:

```
// Acesta este un comentariu
```

Când compilatorul de C întâlneşte cele două caractere slash, el ignoră textul care urmează până la sfârşitul liniei curente. Următorul program, *coment.c*, ilustrează modul de inserare a comentariilor:

```
// Program: coment.c
// Scris de: Kris Jamsa si Lars Klander
// Data scrierii: 22-12-97
// Scop: Ilustrarea utilizarii comentariilor intr-un
// program in C.
#include <stdio.h>
void main(void)
  {
    printf("Totul despre C/C++"); // Afiseaza un mesaj
  }
```

Citind comentariile din acest exemplu, vă puteţi da seama imediat când, cine şi de ce a scris programul. Ar trebui să vă obişnuiţi să plasaţi comentarii similare la începutul programelor dumneavoastră. Dacă alţi programatori care vor să citească sau să modifice programul au de pus întrebări, ei vor afla imediat cine este autorul programului iniţial.

Când compilatorul de C întâlneşte cele două caractere slash, el ignoră textul care urmează pe linia respectivă. În cele mai multe dintre fişierele sursă recente, comentariile sunt introduse cu ajutorul celor două caractere slash. Dacă citiţi un program în C mai vechi, probabil că veţi întâlni comentarii scrise în altă formă. În cadrul acesteia, mesajul apare scris între caractere slash şi asteriscuri, ca mai jos:

```
/* Acesta este un comentariu */
```

Când compilatorul întâlnește simbolul de început de comentariu (/*), el ignoră tot textul care urmează, până la simbolul de încheiere a comentariului (*/). Folosind formatul */* comentariu */*, un mesaj poate apărea pe două sau mai multe linii. Următorul program, *coment2.c*, ilustrează modul de folosire a formatului */* comentariu */*:

```
/* Program: coment2.c
   Scris de: Kris Jamsa si Lars Klander
   Data scrierii: 08-22-97

   Scop: Ilustrarea utilizarii comentariilor intr-un program
   in C. */

#include <stdio.h>

void main(void)
  {
    printf("Totul despre C/C++"); /* Afiseaza un mesaj */
}
```

Așa cum puteți vedea, primul comentariu al programului conține cinci linii. Atunci când folosiți formatul */* comentariu */* pentru comentariile dumneavoastră, aveți grijă ca fiecărui simbol de început de comentariu (/*) să-i corespundă unul de final (*/). Dacă simbolul de final lipsește, compilatorul de C va ignora o mare parte a programului, ceea ce va produce erori de sintaxă dificil de detectat.

Cele mai multe compilatoare de C vor semnala o eroare de sintaxă dacă încercați să plasați un comentariu în interiorul altuia (comentarii imbricate), așa cum prezentăm mai jos:

```
/* Acest comentariu contine un /* al doilea */ comentariu */
```

17 ÎMBUNĂTĂȚIREA LIZIBILITĂȚII PROGRAMULUI C/C++

În secțiunea 16, ați învățat cum să folosiți comentariile în interiorul programelor dumneavoastră pentru a le face mai ușor de citit. De fiecare dată când creați un program, gândiți-vă că va veni o vreme când cineva (dumneavoastră sau alt programator) va trebui să modifice ceva în el. De aceea, este esențial să vă scrieți programele astfel încât să fie ușor de citit. Următorul program în C, *citigreu.c*, va afișa un mesaj pe ecranul dumneavoastră:

```
#include <stdio.h>
void main(void){printf("Totul despre C/C++");}
```

Cu toate că se va compila și va afișa cu succes mesajul dorit, programul este dificil de citit, în cel mai bun caz. Un program bun nu numai că funcționează, dar este, în același timp, ușor de citit și de înțeles. Cheia pentru a crea programe lizibile este includerea de comentarii care să explice ce se întâmplă în program și folosirea liniilor goale pentru a da o formă inteligibilă programului. În următoarele capitole veți învăța despre rolul important pe care îl joacă indentarea pentru a face cât mai lizibil fișierul sursă.

MESAJELE DE AVERTIZARE

C/C++ 18

Atunci când programul conține una sau mai multe erori de sintaxă, compilatorul de C va afișa mesaje de eroare pe ecran și nu va crea un program executabil. Este posibil ca uneori compilatorul să afișeze unul sau mai multe mesaje de avertizare pe ecran, dar totuși să creeze fișierul executabil. De exemplu, următorul program în C, *nu_stdio.c*, nu include fișierul antet *stdio.h*:

```
void main(void)
   {
     printf("Totul despre C/C++");
   }
```

Atunci când compilați acest program, compilatorul *Turbo C++ Lite* va afișa următorul mesaj de avertizare:

```
Warning nu_stdio.c 3: Function 'printf' should have a prototype
in function main().
```

Atunci când compilatorul afișează un avertisment, ar trebui să determinați imediat cauza atenționării și să o corectați. Chiar dacă situațiile semnalate de avertismente nu vor cauza erori în timpul execuției programului, ele pot crea condiții pentru apariția mai târziu a unor erori greu de corectat. Localizând și corectând cauzele care au generat avertismentele compilatorului, veți învăța mult mai mult despre mecanismele interne ale limbajelor C și C++.

CONTROLUL AVERTISMENTELOR COMPILATORULUI

C/C++ 19

În secțiunea 18, ați învățat că trebuie să acordați atenție mesajelor de avertizare pe care compilatorul vi le afișează pe ecran. Pentru a vă ajuta să folosiți mai bine aceste mesaje de avertizare, multe compilatoare vă permit să stabiliți nivelul dorit al mesajelor. În funcție de compilatorul dumneavoastră, puteți folosi o opțiune în linia de comandă pentru a controla nivelul avertismentelor sau puteți folosi directive *pragma*, așa cum se arată în secțiunea 145. O directivă *pragma* este destinată compilatorului. Așa cum veți învăța mai târziu, fiecare compilator suportă directive *pragma* diferite. De exemplu, pentru a dezactiva avertismentul *Identifier is declared but never used* din *Turbo C++ Lite*, codul dumneavoastră trebuie să includă următoarea directivă *pragma*:

```
#pragma warn -use
```

Dacă nu utilizați *Turbo C++ Lite*, consultați documentația care însoțește compilatorul pentru a detemina cum puteți dezactiva anumite mesaje de avertizare.

FOLOSIREA COMENTARIILOR PENTRU A EXCLUDE INSTRUCȚIUNILE DIN PROGRAM

C/C++ 20

În secțiunea 16, ați învățat că puteți să utilizați comentarii în interiorul programelor dumneavoastră pentru a le face mai inteligibile. Când veți crea programe mai complexe, veți putea folosi comentariile pentru a corecta erorile. Când compilatorul de C întâlnește cele două caractere slash (//), ignoră tot textul care urmează pe linia curentă. De asemenea, când un compilator întâlnește simbolul de început de comentariu (/*), ignoră tot textul care

urmează până la simbolul de încheiere a comentariului (*/). În timp ce testați programele, de multe ori veți dori să eliminați una sau mai multe instrucțiuni. O metodă de a elimina instrucțiuni din program este pur și simplu ștergerea lor din fișierul sursă. O a doua metodă de eliminare a instrucțiunilor este transformarea lor în comentariu. Următorul program, *nu_apare.c*, transformă în comentariu toate instrucțiunile *printf*.

```
#include <stdio.h>

void main(void)
  {
    // printf("Aceasta linie nu apare.");
    /* Acesta este un comentariu
    printf("Nici aceasta linie nu apare.");
    */
  }
```

Deoarece ambele instrucțiuni *printf* apar în interiorul comentariilor, compilatorul le ignoră pe amândouă. Ca rezultat, programul nu va afișa nimic la executare. Pe măsură ce programele vor deveni mai complexe, va fi foarte convenabilă folosirea comentariilor pentru a dezactiva instrucțiunile.

Așa cum ați citit în secțiunea 16, cele mai multe compilatoare de C vor returna una sau mai multe erori de sintaxă dacă veți încerca să plasați un comentariu în interiorul altuia. Când utilizați comentarii pentru a dezactiva instrucțiuni, aveți grijă să nu le imbricați.

21 IMPORTANȚA PAGINILOR C/C++

Pe măsură ce veți parcurge secțiunile acestei cărți, veți întâlni variabile și funcții al căror nume începe cu un caracter de subliniere (_), cum ar fi *_dos_getdrive* sau *_chmod*. De obicei, veți utiliza asemenea variabile și funcții numai în mediul DOS. Dacă scrieți programe care se vor executa sub DOS, Windows, Macintosh, UNIX sau alte sisteme de operare, ar trebui să aveți grijă când folosiți aceste funcții, deoarece este posibil ca ele să nu fie disponibile în celelalte sisteme. De aceea, pentru a muta programele din DOS în alt sistem de operare, va fi nevoie să mai lucrați la program. Unele funcții pot avea două implementări; una care începe cu un caracter de subliniere (*_chmod*), iar una fără (*chmod*). Ca regulă, folosiți funcția sau variabila care nu utilizează caracterul de subliniere (în acest caz, *chmod*).

22 CARACTERUL PUNCT ȘI VIRGULĂ C/C++

Examinând diferite programe scrise în C, veți vedea că este folosit foarte mult caracterul punct și virgulă (;). Acest caracter are o importanță deosebită în cadrul limbajului C. Așa cum știți, un program este o listă de instrucțiuni pe care doriți să le execute calculatorul. Când specificați aceste instrucțiuni în C, folosiți caracterul punct și virgula pentru a separa o instrucțiune de alta. Pe măsură ce programele vor deveni mai complexe, veți vedea că sunt situații în care o instrucțiune nu încape pe o singură linie. Atunci când compilatorul examinează programul, el utilizează punctul și virgula pentru a distinge o instrucțiune de următoarea. Sintaxa limbajului C stabilește utilizarea caracterului punct și virgulă. Dacă omiteți acest caracter, va apărea o eroare de sintaxă și programul nu se va compila cu succes.

PREZENTAREA VARIABILELOR

C/C++ 23

Pentru a realiza lucruri folositoare, programele trebuie să stocheze informația, cum ar fi un document pe care îl editați în mai multe sesiuni de lucru cu calculatorul, într-un fișier și în memoria internă. După cum știți, de fiecare dată când rulați un program, sistemul de operare încarcă instrucțiunile programului în memoria calculatorului. Pe măsură ce programul rulează, el stochează valori în locații de memorie. De exemplu, să presupunem că aveți un program care tipărește un document. De fiecare dată când rulați programul, el va afișa un mesaj care vă cere numele fișierului și numărul de copii pe care doriți să le tipăriți. Când scrieți aceaste informații, programul păstrează valorile pe care le-ați introdus în anumite locații de memorie. Pentru a ajuta programul să găsească locația de memorie unde au fost plasate datele, fiecare locație de memorie are o *adresă* unică, cum ar fi locația 0, 1, 2, 3 și așa mai departe. Deoarece pot fi milioane de asemenea adrese, ținerea evidenței fiecărei locații poate deveni foarte dificilă. Pentru a simplifica stocarea informației, programele definesc *variabile*. Acestea sunt niște nume pe care programul le asociază cu anumite locații din memorie. Așa cum indică și cuvântul variabilă, *valoarea* pe care programul o păstrează în aceste locații se poate modifica în cursul execuției programului.

Fiecare variabilă este de un anumit *tip*, care prezintă calculatorului cât de multă memorie necesită datele pe care variabila le păstrează și ce operații poate îndeplini programul cu aceste date. În exemplul anterior, programul care tipărește un document ar trebui să utilizeze o variabilă denumită *nume_fisier* (care păstrează numele fișierului pe care doriți să-l tipăriți) și una denumită *nr_copii* (care păstrează numărul de copii pe care doriți să le tipăriți). În interiorul programului, vă veți referi la variabile după numele lor. De aceea, ar trebui să dați nume semnificative fiecărei variabile. În interiorul programelor în C, veți declara de obicei variabilele imediat după *main*, înainte de instrucțiunile programului, ca mai jos:

```
Void main(void)
  {
      // Aici e locul variabilelor
    printf("Totul despre C/C++")
  }
```

Următorul program vă arată cum ar trebui să declarați trei variabile întregi (variabile care păstrează numere întregi, cum ar fi 1, 2 și 3):

```
void main(void)
  {
    int varsta;      // Varsta utilizatorului in ani
    int greutate;    // Greutatea utilizatorului in kg
    int inaltime;    // Inaltimea utilizatorului in cm
  // Alte instructiuni vor fi scrise aici
  }
```

Fiecare variabilă este de un anumit tip, care definește cantitatea de memorie de care are nevoie variabila, ca și operațiile pe care programul le poate îndeplini cu aceste date. Pentru a declara o variabilă întreagă, programele în C utilizează tipul *int*. După ce declarați o variabilă (ceea ce înseamnă să indicați programului numele variabilei și tipul ei), puteți să atribuiți acestei variabile o valoare (cu alte cuvinte, puteți să stocați informație în ea).

24 Atribuirea de valori variabilelor



O variabilă este un nume pe care programul dumneavoastră îl asociază unei locații de memorie. După ce declarați o variabilă într-un program, îi puteți atribui o valoare. În C se atribuie o valoare unei variabile utilizând semnul egal (=), denumit *operator de atribuire*. Următorul program declară trei variabile de tipul *int* și apoi atribuie fiecărei variabile o valoare:

```
void main(void)
  {
    int varsta;        // Varsta utilizatorului in ani
    int greutate;      // Greutatea utilizatorului in kg
    int inaltime;      // Inaltimea utilizatorului in cm

    varsta = 41;       // Atribuirea varstei utilizatorului
    greutate = 75;     // Atribuirea greutatii utilizatorului
    inaltime = 180;    // Atribuirea inaltimii utilizatorului

      // Alte instructiuni ale programului
  }
```

25 Tipurile variabilelor



Atunci când declarați variabile într-un program, trebuie să indicați compilatorului numele și tipul variabilei. Un tip definește mulțimea de valori pe care variabila poate să le păstreze, ca și setul de operații pe care programul poate să le realizeze cu aceste date. Limbajul C acceptă patru tipuri de bază. Fiecare dintre ele apare în tabelul 25:

Numele tipului	**Caracteristici**
char	Păstrează un singur caracter, cum ar fi literele de la A la Z.
int	Păstrează numere întregi, cum ar fi 1, 2 și 3, precum și numere negative.
float	Păstrează numere reale în format cu virgulă mobilă, în simplă precizie, cum ar fi 3,14 sau –54,1343
double	Păstrează numere reale în format cu virgulă mobilă, în dublă precizie (care este mult mai exact decât formatul cu simplă precizie). Veți utiliza *double* pentru numere foarte mari sau foarte mici.

Tabelul 25 *Cele patru tipuri de bază acceptate de limbajul C.*

Pe parcursul cărții, aceste tipuri sunt examinate în detaliu. Cele mai multe dintre secțiunile cărții vor utiliza una sau mai multe variabile din aceste tipuri de bază.

26 Declararea mai multor variabile de același tip



Așa cum ați învățat în secțiunea 24, atunci când declarați o variabilă într-un program, trebuie să-i indicați compilatorului numele și tipul variabilei. Următoarele instrucțiuni declară trei variabile de tipul *int*:

```
int varsta;
int greutate;
int inaltime;
```

Atunci când declarați variabile de același tip, limbajul C vă permite să înșiruiți numele variabilelor pe una sau mai multe linii, utilizând virgula ca separator între numele lor, ca mai jos:

```
int varsta, greutate, inaltime;
float salariu, taxe;
```

COMENTAREA VARIABILELOR LA DECLARARE

C/C++ 27

În programele în C, comentariile îl ajută pe cel care citește să înțeleagă mai bine programul. Atunci când alegeți numele variabilelor, trebuie să aveți grijă ca acestea să descrie semnificativ valoarea pe care o va păstra variabila. De exemplu, să considerăm următoarele declarații:

```
int varsta, greutate, inaltime;
int x, y, z;
```

Ambele declarații creează trei variabile de tipul *int*. În prima declarație, vă puteți face o idee despre cum este utilizată variabila prin simpla examinare a numelui ei. În afară de utilizarea unui nume semnificativ, puteți, de asemenea, să introduceți explicații suplimentare plasând un comentariu în vecinătatea fiecărei declarații de variabilă, ca mai jos:

```
int varsta;             // Varsta utilizatorului in ani
int greutate;           // Greutatea utilizatorului in kg
int inaltime;           // Inaltimea utilizatorului in cm
```

ATRIBUIREA DE VALORI LA DECLARAREA VARIABILELOR

C/C++ 28

După ce declarați o variabilă într-un program, puteți utiliza *operatorul de atribuire* al limbajului C (semnul egal) pentru a atribui o valoare variabilei. Limbajul C vă permite să atribuiți o valoare variabilei chiar în cadrul declarației. Procesul prin care se atribuie variabilei prima sa valoare este numit de programatori *inițializarea* variabilei. Următoarele instrucțiuni, de exemplu, declară și inițializează trei variabile de tipul *int*:

```
int varsta = 41;      // Varsta utilizatorului in ani
int greutate = 75;    // Greutatea utilizatorului in kg
int inaltime = 180;   // Inaltimea utilizatorului in cm
```

INIȚIALIZAREA MAI MULTOR VARIABILE ÎN CURSUL DECLARAȚIEI

C/C++ 29

În secțiunea 26, ați învățat că limbajul C vă permite să declarați două sau mai multe variabile pe aceeași linie, ca mai jos:

```
int varsta, greutate, inaltime;
```

Atunci când declarați mai multe variabile pe aceeași linie, limbajul C vă permite să inițializați una sau mai multe dintre ele.

```
int varsta = 41, greutate, inaltime = 180;
```

În acest exemplu, se vor inițializa variabilele *varsta* și *inaltime*, iar variabila *greutate* rămâne neinițializată.

30 ALEGEREA UNOR NUME SEMNIFICATIVE DE VARIABILE



Atunci când declarați variabile în programul dumneavoastră, trebuie să alegeți nume semnificative, care descriu modul de utilizare al variabilei. Puteți utiliza o combinație de litere mari și mici. Așa cum ați văzut în secțiunea 8, compilatorul de C face deosebire între literele mari și cele mici. De aceea, dacă alegeți un nume de variabilă alcătuit din litere mari și mici, trebuie să folosiți întotdeauna aceeași combinație. Pentru început, probabil că vă veți limita la folosirea literelor mici, pentru că în acest mod reduceți posibilitatea erorilor provenite din folosirea unui amestec de litere mari și mici.

Trebuie să-i dați un nume unic fiecărei variabile pe care o declarați într-un program. În general, puteți utiliza un număr nelimitat de caractere în numele unei variabile. Numele variabilelor pot conține o combinație de litere, numere sau caractere de subliniere (_), dar numele trebuie să înceapă numai cu o literă sau un caracter de subliniere (_). Următorul fragment de cod prezintă câteva nume valide de variabile:

```
int ore_munca;
float procent_impozit;
float _taxe_6_luni;  // Numele inceput cu caracter de
                     // subliniere e valid
```

Limbajul C are câteva cuvinte cheie predefinite, care au o semnificație specială pentru compilator. Un *cuvânt cheie* este un cuvânt care are înțeles pentru compilator, fără ca dumneavoastră să-i fi dat acest înțeles. De exemplu, *float, int* și *char* sunt cuvinte cheie. Numele de variabile nu pot utiliza aceste cuvinte cheie. Secțiunea 31 prezintă cuvintele cheie ale limbajului C.

31 CUVINTELE CHEIE ALE LIMBAJULUI C



Limbajul de programare C definește câteva cuvinte cheie care au un înțeles special pentru compilator. Când alegeți numele variabilelor (sau creați propriile dumneavoastră funcții), nu utilizați aceste cuvinte cheie. Tabelul 31 listează cuvintele cheie ale limbajului C.

Cuvintele cheie ale limbajului C

auto	default	*float*	*register*	struct	*volatile*
break	do	*for*	*return*	switch	*while*
case	double	*goto*	*short*	typedef	
char	else	*if*	*signed*	union	
const	enum	*int*	*sizeof*	unsigned	
continue	extern	long	static	void	

Tabelul 31 *Lista cuvintelor cheie ale limbajului C.*

VARIABILELE DE TIP INT

C/C++ 32

O *variabilă* este un nume pe care compilatorul de C îl asociază cu una sau mai multe locații de memorie. Atunci când declarați o variabilă într-un program, trebuie să specificați tipul și numele ei. *Tipul* unei variabile specifică genul de valori pe care variabila le poate păstra și setul de operații pe care programul le poate executa asupra datelor. Compilatorul de C alocă de obicei 16 biți (2 octeți) pentru a păstra valori de tipul *int*. Variabilele de tip *int* pot să păstreze valori în intervalul dintre- 32768 și 32767. Figura 32 arată cum este reprezentată o variabilă de tip număr întreg în limbajul C:

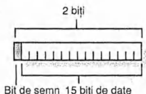


Figura 32 *Reprezentarea în C a unei valori întregi.*

Valorile de tip *int* sunt numere întregi. Ele nu au o parte fracționară, așa cum au numerele reale (reprezentate în virgulă mobilă). Dacă dați o valoare reală unei variabile de tip *int*, multe compilatoare de C, pur și simplu vor ignora partea fracționară. Dacă îi dați unei variabile de tip *int* o valoare în afara intervalului dintre –32768 și 32767, atunci va apărea o situație de depășire (overflow), iar valoarea atribuită va fi eronată.

VARIABILELE DE TIP CHAR

C/C++ 33

O *variabilă* este un nume pe care compilatorul de C îl asociază cu una sau mai multe locații de memorie. Atunci când declarați o variabilă într-un program, trebuie să specificați tipul și numele ei. *Tipul* unei variabile specifică genul de valori pe care variabila le poate păstra și setul de operații pe care programul le poate executa asupra datelor. Limbajul C folosește tipul *char* pentru a păstra valori de tip caracter. Compilatorul de C alocă, de obicei 8 biți (1 octet) pentru a păstra valori de tip *char*. O variabilă de tip *char* poate păstra numere întregi cu valori cuprinse între –128 și 127. Figura 33 arată cum reprezintă limbajul C o valoare de tip *char*.

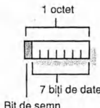


Figura 33 *Reprezentarea în C a unei valori de tip* ***char***.

Programele pot atribui o valoare unei varibile de tip *char* într-una din cele două modalități prezentate în continuare. Prima modalitate constă în acordarea valorii ASCII corespunzătoare caracterului. De exemplu, litera A are valoarea ASCII 65:

```
char litera = 65; // Variabilei litera i se atribuie caracterul A
```

A doua modalitate este cea în care programul folosește o constantă de tip caracter care apare între două apostrofuri ('):

```
char litera = 'A';
```

Variabilele de tip *char* nu pot păstra decât o literă la un moment dat. Pentru a păstra mai multe caractere, trebuie să declarați un șir de caractere, care va fi explicat în secțiunea „Șirurile".

34 VARIABILELE DE TIP FLOAT

C/C++

O *variabilă* este un nume pe care compilatorul de C îl asociază cu una sau mai multe locații de memorie. Atunci când declarați o variabilă într-un program, trebuie să specificați tipul și numele ei. *Tipul* unei variabile specifică genul de valori pe care variabila le poate păstra și setul de operații pe care programul le poate executa asupra datelor. Limbajul C folosește tipul *float* pentru a păstra valori reale în virgulă mobilă (numere negative și pozitive care conțin părți fracționare). Compilatorul de C va aloca, de obicei, 32 de biți (4 octeți) pentru a păstra valori de tip *float*. O variabilă de tip *float* poate păstra valori cu precizie de șase sau șapte cifre în intervalul dintre 3.4E–38 și 3.4E+38.

Limbajul C păstrează valoarea ca o *mantisă* pe 23 de biți (care conține partea fracțională), un exponent pe 8 biți (care conține puterea la care calculatorul ridică numărul când îi calculează valoarea) și un singur bit de semn (care determină dacă valoarea este pozitivă sau negativă). Cu alte cuvinte, dacă o variabilă conține valoarea 3.4E+38, bitul de semn va fi 0, ceea ce indică un număr pozitiv, mantisa de 23 de biți va conține o reprezentare binară a numărului 3,4, iar exponentul pe 8 biți va conține o reprezentare binară a exponentului 10^{38}. Figura 34 ilustrează modalitatea de reprezentare a unei variabile de tipul *float*. Secțiunea 337 va explica în detaliu ce sunt mantisa și exponentul.

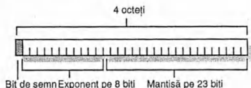


Figura 34 *Reprezentarea în C a unei valori de tip* ***float.***

Observație: *Această secțiune, ca și altele care urmează, reprezintă numerele reale folosind* ***notația științifică.*** *Această notație vă permite să reprezentați orice număr utilizând o singură cifră în stânga virgulei, un număr nelimitat de cifre în dreapta virgulei și un exponent reprezentând o putere a lui 10. Când determinați valoarea reală a numărului, înmulțiți numărul (**mantisa**) cu valoarea 10 la puterea x (unde **x** reprezintă **exponentul**). De exemplu, numărul 3.1415967E+7 se evaluează ca 31415967.0 sau* $3.1415967 \cdot 10^7$.

VARIABILELE DE TIP DOUBLE

C/C++ 35

O *variabilă* este un nume pe care compilatorul de C îl asociază cu una sau mai multe locații de memorie. Atunci când declarați o variabilă într-un program, trebuie să specificați tipul și numele ei. *Tipul* unei variabile specifică genul de valori pe care variabila le poate păstra și setul de operații pe care programul le poate executa asupra datelor. Limbajul C folosește tipul *double* pentru a păstra valori reale în virgulă mobilă (numere negative și pozitive care conțin părți fracționare). Compilatorul de C va aloca, de obicei, 64 de biți (8 octeți) pentru a păstra valori de tip *double*. O variabilă de tip *double* poate păstra valori cu precizie de 14 sau 15 cifre în intervalul dintre 1.7E-308 și 1.7E+308. Figura 35 ilustrează modalitatea de reprezentare a unei variabile de tipul *double*.

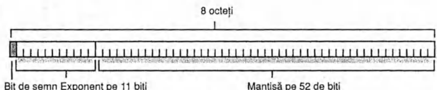


Figura 35 *Reprezentarea în C a unei valori de tip* ***double.***

ATRIBUIREA DE VALORI VARIABILELOR ÎN VIRGULĂ MOBILĂ

C/C++ 36

O valoare reală în *virgulă mobilă* este o valoare care conține o parte fracționară, cum ar fi 123.45. Atunci când utilizați într-un program valori în virgulă mobilă, vă puteți referi la ele folosind formatul zecimal, cum ar fi 123.45, sau puteți folosi formatul exponențial al acestei valori, 1.2345E2. Deci, ambele instrucțiuni din următoarea secvență de cod acordă variabilei *raza* aceeași valoare:

```
raza = 123.45;
raza = 1.2345E2;
```

În mod similar, ambele instrucțiuni de mai jos atribuie variabilei *raza* aceeași valoare:

```
raza = 0.12345;
raza = 12.345E-2;
```

MODIFICATORII DE TIP

C/C++ 37

Limbajul C oferă patru tipuri principale de date (*int, char, float* și *double*). Așa cum ați învățat, fiecare tip definește un set de valori pe care variabila le poate păstra și un set de operații pe care programul le poate executa cu aceste date. De asemenea, ați învățat că variabilele de tip *int* pot păstra valori în intervalul –32768 până la 32767, iar variabilele de tip *char* pot păstra valori în intervalul –128 până la 127. Pentru a vă ajuta să modificați intervalul valorilor pe care variabilele de tip *int* și *char* pot să le păstreze, limbajul C furnizează un set de modificatori de tip: *unsigned, long, register, signed* și *short*. Un *modificator de tip* schimbă domeniul valorilor pe care o variabilă le poate păstra sau modul în care compilatorul

păstrează o variabilă. Pentru a modifica un tip, plasați modificatorul în fața numelui tipului, în declarația variabilei, ca mai jos:

```
unsigned int total_inventar;
register int contor;
long int numar_foarte_mare;
```

Următoarele câteva secțiuni analizează în detaliu acești modificatori de tip.

38 MODIFICATORUL DE TIP UNSIGNED

C/C++

Un *modificator de tip* schimbă domeniul valorilor pe care o variabilă le poate păstra sau modul în care compilatorul păstrează o variabilă. Așa cum ați învățat, variabilele de tip *int* pot păstra valori în intervalul –32768 până la 32767. În reprezentarea unei valori de tip *int*, cel mai semnificativ bit al valorii indică semnul acesteia (pozitiv sau negativ), așa cum ați văzut în secțiunea 32. În unele cazuri, programul dumneavoastră nu atribuie niciodată o valoare negativă unei anumite variabile. Modificatorul de tip *unsigned* îi indică compilatorului să nu folosească cel mai semnificativ bit ca bit de semn, ci să-l folosească pentru a reprezenta valori pozitive mai mari. O variabilă de tip *unsigned int* poate avea valori în intervalul 0-65535. Figura 38.1 indică modul în care compilatorul de C reprezintă o variabilă de tipul *unsigned int*.



Figura 38.1 *Reprezentarea în C a unei valori de tip* ***unsigned int.***

Așa cum s-a arătat în secțiunea 33, variabilele de tip *char* pot pastra valori în intervalul dintre –128 și 127. Atunci când folosiți modificatorul de tip *unsigned* cu variabile de tip *char*, puteți crea variabile care păstrează valori în intervalul 0-255. Figura 38.2 ilustrează modul în care compilatorul de C reprezintă o variabilă de tip *unsigned char*.

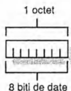


Figura 38.2 *Reprezentarea în C a unei valori de tip* ***unsigned char.***

Următoarea secvență prezintă declarații de variabile de tipul *unsigned int* sau *unsigned char*.

```
void main(void)
  {
```

```
    unsigned int secunde_curente;
   unsigned int indicator_stare;
   unsigned char chenar_meniu;    // Caracter din setul ASCII
                                  // extins
}
```

MODIFICATORUL DE TIP LONG

C/C++ 39

Un modificator de tip schimbă domeniul valorilor pe care o variabilă le poate păstra sau modul în care compilatorul păstrează o variabilă. Variabilele de tipul *int* pot păstra valori pozitive și negative în intervalul –32768 până la 32767. Cum s-a arătat anterior, în secțiunea 32, compilatorul de C reprezintă valori de tip *int* folosind 16 biți, cel mai semnificativ bit indicând semnul valorii. În cele mai multe cazuri, programele dumneavoastră trebuie să păstreze numere întregi care sunt mai mari (peste 32768) sau mai mici (sub –32768) decât valorile pe care le poate păstra o variabilă de tipul *int*. Modificatorul de tip *long* indică compilatorului să utilizeze 32 de biți (4 octeți) pentru a reprezenta un număr întreg. Variabila de tip *long* poate să păstreze valori în intervalul dintre- 2147483648 și 2147483647. Figura 39 arată cum păstrează compilatorul de C o variabilă de tipul *long int*.



Figura 39 *Reprezentarea în C a unei valori de tip* ***long int.***

Observație: *Multe compilatoare de C++ acceptă și tipul* ***long double****, prin care programul dumneavoastră poate să folosească numere în virgulă mobilă cu precizie de până la 80 de cifre, mai mult decât obișnuita precizie de 64 de cifre. Valorile de tip* ***long double*** *folosesc 10 octeți de memorie, din care 60 de biți pentru mantisă și 19 biți pentru exponent. Intervalul pentru valorile long double este de la 3.4E–4932 la 1.1E+4932. Pentru a stabili dacă aveți un compilator care acceptă declararea variabilelor* ***long double****, consultați documentația compilatorului.*

UTILIZAREA COMBINATĂ A MODIFICATORILOR DE TIP UNSIGNED ȘI LONG

C/C++ 40

În secțiunea 38, ați învățat că modificatorul de tip *unsigned* cere compilatorului de C să nu interpreteze valoarea celui mai semnificativ bit ca indicator de semn, ci să folosească acest bit pentru reprezentarea unor valori mai mari. De asemenea, în secțiunea 39 ați învățat că modificatorul de tip *long* cere compilatorului să dubleze numărul de biți pe care îi folosește pentru a reprezenta valori întregi. În anumite cazuri, programele dumneavoastră pot avea nevoie să păstreze valori pozitive foarte mari. Combinând utilizarea modificatorilor de tip *unsigned* și *long*, puteți să indicați compilatorului dumneavoastră să aloce 32 de biți pentru variabile cu valori cuprinse în intervalul de la 0 la 4292967265. Figura 40 ilustrează modul în care compilatorul de C reprezintă o variabilă de tipul *unsigned long int*.

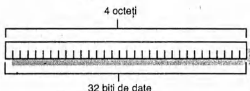


***Figura 40** Reprezentarea în C a unei valori de tip **unsigned long int**.*

Exemplul următor ilustrează declararea unor variabile de tip *unsigned long int:*

```
void main(void)
  {
    unsigned long int valoare_foarte_mare;
    unsigned long int datorie_externa;
  }
```

41 UTILIZAREA UNOR VALORI MARI C/C++

Aşa cum aţi învăţat, variabilele de tip *int* pot păstra valori în intervalul –32768 până la 32767. De asemenea, variabilele de tip *long int* pot păstra valori în intervalul de la –2147483648 până la 2147483647. Când lucraţi cu valori mari în programele dumneavoastră, nu includeţi virgula de separare (punctul de separare din sistemul european de numerotare - *n.t.*). Lucraţi cu aceste numere ca în exemplul de mai jos:

```
long int numar_mare = 1234567;
long int un_milion = 1000000;
```

Dacă includeţi virgula separatoare în interiorul numărului, compilatorul de C va genera o eroare de sintaxă.

42 MODIFICATORUL DE TIP REGISTER C/C++

O variabilă este un nume pe care programul îl asociază cu o locaţie de memorie. Atunci când declaraţi o variabilă, compilatorul de C alocă memorie pentru a păstra valoarea variabilei. Când programul dumneavoastră trebuie să acceseze variabila, apare o mică suprasarcină (calculatorul consumă un anumit interval de timp) datorită accesării memoriei de către CPU (unitatea centrala de prelucrare). În funcţie de utilizarea variabilei, puteţi să indicaţi uneori calculatorului să păstreze variabila într-un registru (care se găseşte chiar în interiorul cipului CPU) pentru a face programul mai performant. Deoarece compilatorul poate să acceseze valoarea mult mai repede atunci când ea este localizată într-un registru, programul dumneavoastră se va executa mai repede. Modificatorul de tip *register* cere compilatorului să păstreze variabila într-un registru, cât mai frecvent posibil. Pentru că CPU are un număr limitat de registre, compilatorul nu poate să asocieze permanent o variabilă unui registru. În schimb, poate să încerce să ţină variabila cât mai mult timp într-un registru. Următoarea secvenţă vă arată cum se foloseşte modificatorul de tip *register*.

```
void main(void)
  {
    register int total;
    register unsigned semnal_stare;
  }
```

Puteți să folosiți modificatorul de tip *register* pentru variabilele pe care programul dumneavoastră le accesează frecvent, cum ar fi o *variabilă de ciclare*, pe care programul o accesează la fiecare repetare a unei bucle.

MODIFICATORUL DE TIP SHORT

C/C++ 43

Cum am explicat în secțiunea 32, compilatorul de C reprezintă de obicei variabilele de tip *int* folosind 16 biți. Ca urmare, variabilele de tip *int* pot păstra valori în intervalul -32768 până la 32767. Totuși, dacă folosiți un compilator pe 32 de biți, acesta poate reprezenta valorile întregi folosind 32 biți, ceea ce înseamnă că o variabilă de tip *int* poate să păstreze valori în intervalul de la -2147483648 până la 2147483647. Dacă folosiți valori în afara intervalului acceptat de variabila de tip *int*, apare o *situație de depășire*, iar valoarea asociată va fi eronată. (Secțiunea 50 explică în detaliu situația de depășire.) Programatorii scriu unele programe știind că, atunci când apare o *depășire*, compilatorul înlocuiește valoarea depășită cu o anumită valoare, care este întotdeauna aceeași. Cu alte cuvinte, programatorul scrie programul pentru a folosi *depășirea*. Dacă ați avea un program care folosește valori de tip *int* în acest mod (adică un program care contează pe valori depășite) și l-ați muta dintr-un mediu pe 16 biți într-unul pe 32 de biți, situația de depășire nu s-ar mai produce, deoarece variabila de tip întreg pe 32 de biți poate păstra valori mai mari. Dacă scrieți un program bazându-vă pe *depășire*, ceea ce presupune ca variabilele de tip *int* să fie reprezentate de compilator pe 16 biți, puteți să folosiți modificatorul de tip *short* pentru a vă asigura că variabila *int* va fi reprezentată de compilator utilizând 16 biți. Următoarea instrucțiune arată cum se declară o variabilă de tip *short int*:

```
void main(void)
  {
    short int valoare_cheie;
    short int numar_mic;
  }
```

OMISIUNEA LUI INT DIN DECLARAȚIILE MODIFICATE

C/C++ 44

În cadrul acestei secțiuni, ați învățat despre mai mulți modificatori de tip, inclusiv despre *long*, *short* și *unsigned*. Următoarea secvență ilustrează modul de utilizare a acestor modificatori:

```
unsigned int semnal_stare;
short int valoare_mica;
long int numar_foarte_mare;
```

Când folosiți acești trei modificatori, cele mai multe compilatoare vă permit să omiteți particula *int*, așa cum se poate vedea mai jos:

```
unsigned semnal_stare;
short valoare_mica;
long numar_foarte_mare;
```

45 MODIFICATORUL DE TIP SIGNED

C/C++

Așa cum ați învățat în secțiunea 33, compilatoarele de C reprezintă de obicei variabile de tip *char* folosind opt biți, dintre care cel mai semnificativ reprezintă semnul valorii. Ca urmare, variabilele de tipul *char* pot păstra valori în intervalul dintre –128 și 127. În secțiunea 38, ați învățat cum să folosiți modificatorul *unsigned* pentru a cere compilatorului de C să nu interpreteze bitul ca semn, ci să-l folosească pentru a reprezenta valori pozitive mai mari. Folosind modificatorul de tip *unsigned*, variabila de tip *char* poate să păstreze valori în intervalul de la 0 la 255. Dacă folosiți o variabilă de tip *char* și îi atribuiți o valoare din afara intervalului de valori valide, apare o situație de depășire și valoarea pe care calculatorul o atribuie variabilei nu va fi cea pe care dumneavoastră o doriți. În anumite cazuri, totuși, veți scrie intenționat programe în care apar depășiri. Dacă vă hotărâți să mutați un astfel de program într-un alt compilator, care poate reprezenta variabile de tip *unsigned char*, puteți să folosiți modificatorul de tip *signed*, pentru a vă asigura că cel de-al doilea compilator va reprezenta variabilele de tip *char* folosind 7 biți pentru date și un bit pentru semn. Următoarea instrucțiune arată cum se declară o variabilă de tip *signed char*:

```
void main(void)
  {
    signed char valoare_octet;
    signed char optiune_meniu;
  }
```

46 OPERAȚII DE ATRIBUIRE MULTIPLĂ

C/C++

Așa cum ați învățat, limbajul C utilizează semnul egal (=) ca operator de atribuire. În mod obișnuit, programul dumneavoastră în C atribuie valori variabilelor pe linii separate, așa cum se vede mai jos:

```
total = 0;
suma = 0;
valoare = 0;
```

Atunci când doriți să atribuiți aceeași valoare mai multor variabile, limbajul C vă permite să realizați toate atribuirile deodată, ca mai jos:

```
total = suma = valoare = 0;
```

Atunci când compilatorul de C întâlnește o operație de atribuire multiplă, el atribuie valorile de la dreapta la stânga. Ca regulă, utilizați atribuirile multiple numai pentru a inițializa variabilele. Folosirea unor astfel de operații pentru situații mai complexe va micșora

lizibilitatea programului dumneavoastră. De exemplu, următorul program atribuie celor două variabile majuscula corespunzătoare caracterului introdus de utilizator.

```
litera_salvata = litera = toupper(getchar());
```

ATRIBUIREA VALORII UNEI VARIABILE DE UN ANUMIT TIP UNEI VARIABILE DE ALT TIP

C/C++ 47

Un tip definește mulțimea de valori pe care o variabilă le poate avea și setul de operații pe care programul le poate executa asupra lor. Limbajul C pune la dispoziție patru tipuri principale de date (*int*, *char*, *float* și *double*). În unele cazuri, veți avea nevoie să atribuiți valoarea unei variabile de tip *int* unei valori de tip *float* sau viceversa. Ca regulă generală, puteți atribui fără probleme o valoare de tipul *int* unei variabile de tipul *float*. Dar atunci când atribuiți valoarea unei variabile de tipul *float* unei variabile de tip *int*, trebuie să fiți precaut. Cele mai multe compilatoare vor trunchia partea zecimală, dezactivând partea fracționară. Pe de altă parte, unele compilatoare ar putea rotunji valoarea în loc de a o trunchia (aceasta însemnând că dacă partea fracționară a valorii este mai mare decât 0,5, cele două compilatoare vor converti valoarea în mod diferit). Dacă doriți să vă asigurați că programul va realiza cu consecvență trecerea de la valori reale în virgulă mobilă la valori întregi, puteți să apelați la funcțiile *ceil* sau *floor*, prezentate în secțiunea „Funcțiile matematice".

CREAREA UNOR TIPURI PROPRII

C/C++ 48

Un tip definește mulțimea de valori pe care o variabilă le poate avea și setul de operații pe care programul le poate executa asupra lor. Limbajul C pune la dispoziție patru tipuri principale de date (*int*, *char*, *float* și *double*). Așa cum ați învățat, puteți combina modificatorii de tip pentru a modifica domeniul valorilor pe care le poate avea o variabilă. Pe măsură ce va crește numărul de variabile din program, probabil că veți considera mai convenabil să creați propriul dumneavoastră nume de variabilă, care să prescurteze definiția tipurilor mai des folosite. De exemplu, să considerăm următoarele declarații de tipul *unsigned long int*:

```
unsigned long int secunde_din_ianuarie;
unsigned long int populatia_lumii_2000;
```

Folosind instrucțiunea *typedef* a limbajului C, puteți defini numele de tip *ULINT* care este identic cu tipul *unsingned long int*, ca mai jos:

```
typedef unsigned long int ULINT;
```

După ce ați creat acest nume de tip, îl puteți folosi pentru a defini variabile, ca mai jos:

```
ULINT secunde_din_ianuarie;
ULINT populatia_lumii_2000;
```

Pe măsură ce programele dumneavoastră vor folosi declarații mai complexe de variabile, veți vedea că este mai convenabil să creați noi nume de tip, deoarece acestea pot economisi timpul de scriere și reduc posibilitatea greșelilor.

Observaţie: *În exemplele din această secţiune am scris ULINT cu majuscule, deoarece este mai uşor pentru alţi programatori să identifice tipurile create de dumneavoastră dacă le reprezentaţi în mod diferit faţă de tipurile obişnuite. Puteţi scrie numele tipului folosind ori numai majuscule, ori numai minuscule, ori o combinaţie a amândurora, după cum doriţi. Totuşi, trebuie să fiţi consecvent în privinţa modului de denumire a tipurilor dumneavoastră în mai multe programe sau a mai multor tipuri în cadrul aceluiaşi program.*

49 ATRIBUIREA DE VALORI HEXAZECIMALE SAU OCTALE



În funcţie de aplicaţia dumneavoastră, probabil că uneori veţi avea nevoie să lucraţi cu valori *octale* (în baza 8) sau *hexazecimale* (în baza 16). În aceste cazuri, îi indicaţi compilatorului că vreţi să lucraţi cu valori care nu sunt zecimale. Dacă puneţi un zero înaintea unei valori, cum ar fi 077, compilatorul de C va interpreta valoarea ca octală. În mod similar, dacă o valoare este precedată de 0x, ca de exemplu 0xFF, compilatorul interpretează valoarea ca hexazecimală. Următoarele instrucţiuni ilustrează modul de utilizare a unor constante octale şi hexazecimale:

```
int val_octala = 0227;
int val_hexa = 0xFF0;
```

50 SITUAŢIILE DE DEPĂŞIRE



Aşa cum aţi învăţat, tipul unei variabile defineşte intervalul de valori pe care le poate avea o variabilă, precum şi operaţiile pe care programul le poate executa asupra ei. Variabilele de tipul *int*, de exemplu, pot avea valori în intervalul de la –32768 la 32767. Dacă atribuiţi o valoare în afara acestui interval unei variabile de tip *int*, va apărea o eroare de *depăşire*. Aşa cum aţi învăţat mai înainte, limbajul C foloseşte 16 biţi pentru a reprezenta variabilele de tip *int*. Compilatorul de C utilizează bitul cel mai semnificativ dintre cei 16 pentru a determina semnul variabilei. Dacă cel mai semnificativ bit este 0, valoarea este pozitivă. Dacă el este 1, valoarea este negativă. Deci C utilizează 15 biţi pentru a reprezenta valoarea variabilei. Pentru a înţelege de ce apare *depăşirea*, trebuie să ţineţi seama de modul de reprezentare pe biţi a valorii variabilei. Să considerăm următoarele valori:

```
    0      0000 0000 0000 0000
    1      0000 0000 0000 0001
    2      0000 0000 0000 0010
    3      0000 0000 0000 0011
    4      0000 0000 0000 0100

32765      0111 1111 1111 1101
32766      0111 1111 1111 1110
32767      0111 1111 1111 1111
```

Dacă adăugați 1 la valoarea 32767, v-ați aștepta ca rezultatul să fie 32768, totuși, în C valoarea va deveni –32678.

```
 32767     0111 1111 1111 1111
    +1     0000 0000 0000 0001
---------  -------------------
-32768     1000 0000 0000 0000
```

Următorul program, *depasire.c*, arată cum apare situația de depășire:

```
#include <stdio.h>

void main(void)
  {
    int pozitiv = 32767;
    int negativ = -32768;

    printf("%d + 1 este %d\n", pozitiv, pozitiv+1);
    printf("%d - 1 este %d\n", negativ, negativ-1);
  }
```

Când compilați și executați acest program, pe ecranul dumneavoastră se va afișa următorul mesaj:

```
32767 + 1 este -32768
-32768 - 1 este 32767
C:\>
```

După cum vedeți, adăugând un număr la 32767, obținem un număr negativ, în timp ce scăzând un număr din –32768, obținem un număr pozitiv. Unul dintre neajunsurile depășirii este faptul că adesea nu observați eroarea în programele dumneavoastră, deoarece compilatorul de C nu returnează un mesaj de eroare când apare situația de depășire. Cu alte cuvinte, programul continuă să se execute, în ciuda apariției unei depășiri. Ca rezultat, când vă depanați programul, s-ar putea să găsiți cu greu erorile datorate depășirii.

Observație: *Dacă folosiți compilatorul* ***Turbo C++ Lite*** *ori cele mai multe dintre noile compilatoare (cum sunt* ***Visual C++***® *al firmei Microsoft sau* ***C++ 5.02***® *al firmei Borland), compilatorul vă va atenționa în legătură cu posibilitatea unei probleme de depășire. Compilatorul* ***Turbo C++ Lite*** *vă va da mesajul de avertizare:* ***Constant is long in function main*** *și* ***Conversion may lose significant digits in function main()****, dar va rula totuși programul (și variabila va fi depășită). Ca regulă generală, chiar dacă mesajul de avertizare al compilatorului nu oprește compilarea programului, trebuie să țineți seama de el și să luați măsurile cuvenite.*

PRECIZIA

C/C++ 51

Așa cum ați învățat, calculatorul reprezintă numerele în interiorul său cu ajutorul combinațiilor de 1 și 0 (numere binare). În secțiunea precedentă, ați învățat că fiecare tip are un interval specific de valori datorită faptului că este reprezentat printr-un număr fix de biți. Dacă atribuiți o valoare în afara intervalului, apare o situație de depășire. În cazul valorilor în virgulă mobilă, depășirea duce la un anumit grad de imprecizie. *Precizia* unei valori definește gradul de exactitate al acesteia. De exemplu, valorile de tip *float* asigură utilizarea

a şase sau şapte cifre semnificative. Să presupunem că atribuiţi valoarea 1.234567890 unei variabile de tip *float*. Pentru că tipul *float* pune la dispoziţie numai şapte cifre semnificative, cea mai precisă estimare a valorii va fi 1.23456. Pe de altă parte, valorile de tip *double* prevăd 14 sau 15 cifre semnificative. De aceea, valoarea de tip *double* poate păstra cu exactitate numărul 1.234567890.

Când lucraţi cu numere fracţionare, trebuie să fiţi conştient că valorile sunt reprezentate de calculator printr-un număr fix de biţi. Ca urmare, este imposibil pentru calculator să reprezinte întotdeauna cu exactitate o valoare. De exemplu, calculatorul poate să reprezinte valoarea 0.4 ca 0.3999999, ori pe 0.1 ca 0.099999 şi aşa mai departe. Următorul program, *precis.c*, arată diferenţa între precizia dublă şi cea simplă:

```
#include <stdio.h>
void main(void)
  {
    float exact = 0.123456790987654321;
    double mai_exact = 0.1234567890987654321;

    printf("Valoarea lui float\t %21.19f\n", exact);
    printf("Valoarea lui double\t %21.19f\n", mai_exact);
  }
```

Când compilaţi şi executaţi programul *precis.c*, pe ecranul dumneavoastră va apărea:

```
Valoarea lui float 0.1234567890432815550
Valoarea lui double 0.1234567890987654380
C:\>
```

52 ATRIBUIREA GHILIMELELOR ŞI A ALTOR CARACTERE



Când lucraţi cu variabile de tip *char* sau cu şiruri de caractere, pot apărea situaţii în care trebuie să atribuiţi unei variabile valoarea de ghilimele sau de apostrof. De exemplu, când scrieţi *Jamsa's C/C++ Programmer's Bible*, trebuie să folosiţi de două ori apostroful în şirul de caractere. În asemenea cazuri, trebuie să plasaţi înaintea caracterului apostrof caracterul backslash (\), ca mai jos:

```
char apostrof = '\'';
char ghilimele = '\"';
```

În afară de caracterul apostrof, puteţi utiliza în programul dumneavoastră şi alte caractere speciale, prezentate în tabelul 52. Pentru a face aceasta, plasaţi simbolul caracterului imediat după caracterul backslash. În toate aceste cazuri trebuie să folosiţi literele mici pentru a reprezenta caracterul special.

Caracter	Semnificaţie
\a	Caracter ASCII de atenţionare
\b	Backspace
\f	Avans hârtie
\n	Linie nouă
\r	Retur de car

Caracter	Semnificație
\t	Tabulator orizontal
\v	Tabulator vertical
\\	Backslash
\'	Apostrof
\"	Ghilimele
\?	Semnul întrebării
\nnn	Valoare ASCII în octal
\xnnn	Valoare ASCII în hexazecimal

***Tabelul 52** Caracterele escape definite în C.*

PREZENTAREA FUNCȚIEI PRINTF



Mai multe dintre secțiunile acestei cărți au folosit funcția *printf* pentru a afișa mesaje pe ecran. Când programul dumneavoastră folosește *printf*, datele pe care vreți să le afișați reprezintă *parametrii* sau *argumentele funcției printf*. Următoarea instrucțiune folosește funcția *printf* pentru a afișa pe ecran mesajul *Totul despre C/C++*:

```
printf("Totul despre C/C++");
```

În acest caz, șirul de caractere (literele care apar între cele două ghilimele) constituie singurul argument al funcției *printf*. Când programul dumneavoastră începe să lucreze cu variabile, poate că veți dori să folosiți *printf* pentru a afișa valoarea fiecărei variabile. Funcția *printf* acceptă mai mulți parametri. Primul parametru trebuie să fie întotdeauna un șir de caractere. Parametrii care urmează primului șir de caractere pot fi numere, variabile, expresii (ca de exemplu 3*15) sau chiar alte șiruri de caractere. Atunci când doriți ca *printf* să afișeze o valoare sau o variabilă, trebuie să includeți informația despre tipul variabilei în primul parametru. Pe lângă caractere, în primul parametru puteți să introduceți *specificatori de format*, care arată funcției *printf* cum să afișeze ceilalți parametri. Un specificator de format este reprezentat prin semnul de procent (%) urmat de o literă. De exemplu, pentru a afișa o valoare întreagă, veți folosi *%d* (*d* specifică valorile zecimale). De asemenea, pentru a tipări o valoare în virgulă mobilă, puteți folosi *%f*. Următoarele instrucțiuni ilustrează utilizarea specificatorilor de format în cadrul funcției *printf*:

```
printf("Varsta utilizatorului este %d\n", varsta);
printf("Impozitul pe vanzari este %d\n", pret * 0.07);
printf("Varsta utilizatorului: %d greutate: %d inaltime: %d\n",
  varsta, greutate, inaltime);
```

După cum vedeți, puteți să introduceți în cadrul primului parametru al lui *printf* unul sau mai mulți specificatori de format. Observați că a treia instrucțiune continuă pe rândul următor. Când instrucțiunea dumneavoastră nu încape pe o singură linie, încercați să găsiți un loc bun pentru a rupe rândul (cum ar fi imediat după virgulă) și indentați (trageți spre interiorul paginii) linia următoare. Scopul indentării este îmbunătățirea aspectului vizual al programului dumneavoastră, astfel încât cel care vă citește programul să deducă mai ușor că respectiva linie o continuă pe cea de mai sus. Mai multe dintre capitolele următoare explică în detaliu diferiți specificatori de format ai funcției *printf*.

54 AFIȘAREA VALORILOR DE TIP INT FOLOSIND FUNCȚIA PRINTF



Funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). Pentru a afișa valorile de tip *int* cu *printf*, se folosește specificatorul de format *%d*. Următorul program, *intout.c*, folosește specificatorul de format *%d* pentru a afișa valori și variabile de tip *int*:

```
#include <stdio.h>
void main(void)
  {
    int varsta = 41;
    int greutate = 75;
    int inaltime = 180;

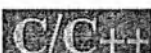
    printf("Varsta utilizatorului: %d greutate:
      %d inaltime: %d\n", varsta, greutate, inaltime);
    printf("%d plus %d egal %d\n", 1, 2, 1 + 2);
  }
```

Atunci când compilați și executați programul *intout.c*, pe ecran se afișează următorul rezultat:

```
Varsta utilizatorului: 41 greutate: 75 inaltime: 180
1 plus 2 egal 3
C:\>
```

Observație: *Multe compilatoare tratează specificatorul de format* ***%i*** *ca fiind identic cu* ***%d****. Dacă vreți să creați un program nou, utilizați totuși specificatorul* ***%d****, pentru că specificatorul* ***%i*** *este un specificator moștenit și este posibil ca viitoarele compilatoare să nu îl accepte.*

55 TIPĂRIREA UNEI VALORI ÎNTREGI OCTALE SAU HEXAZECIMALE



Funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). În funcție de programul dumneavoastră, poate că, la un moment dat, veți dori să afișați o valoare întreagă octală (în baza 8) sau hexazecimală (în baza 16). Caracterul *%o* (litera o, nu cifra 0) este specificatorul care indică funcției *printf* să afișeze valoarea în format octal. Într-un mod similar, specificatorii *%x* și *%X* indică funcției *printf* să afișeze valoarea în format hexazecimal. Diferența între *%x* și *%X* este că cel din urmă afișează valoarea hexazecimală cu majuscule. Următorul program, *oct_hex.c*, ilustrează utilizarea specificatorilor de format *%o*, *%x* și *%X*:

```
#include <stdio.h>
void main(void)
  {
    int valoare = 255;

    printf("Valoarea zecimala %d este %o in octal\n",
      valoare, valoare);
```

```
    printf("Valoarea zecimala %d este %x in hexazecimal\n",
      valoare, valoare);
    printf("Valoarea zecimala %d este %X in hexazecimal\n",
      valoare, valoare);
  }
```

Atunci când compilați și executați programul *oct_hex.c*, pe ecran va fi afișat următorul rezultat:

```
Valoarea zecimala 255 este 377 in octal
Valoarea zecimala 255 este ff in hexazecimal
Valoarea zecimala 255 este FF in hexazecimal
C:\>
```

AFIȘAREA VALORILOR DE TIP UNSIGNED INT FOLOSIND FUNCȚIA PRINTF

C/C++ 56

Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int*, *float*, *char* și așa mai departe). Pentru a afișa valorile de tip *unsigned int* cu funcția *printf*, trebuie să folosiți specificatorul de format *%u*. Dacă folosiți *%d* în loc de *%u*, *printf* va trata valoarea respectivă ca fiind de tip *int* și probabil că va afișa un rezultat greșit. Următorul program, *u_intout.c*, folosește specificatorii de format *%u* și *%d* pentru a afișa valoarea 42000. Programul *u_intout.c* ilustrează tipul de eroare care poate apărea dacă folosiți un specificator de format greșit:

```
#include <stdio.h>
void main(void)
  {
    unsigned int valoare = 42000;

    printf("Afiseaza 42000 ca unsigned %u\n", valoare);
    printf("Afiseaza 42000 ca int %d\n", valoare);
  }
```

Atunci când compilați și executați programul *u_intout.c*, pe ecran va apărea următorul rezultat:

```
Afiseaza 42000 ca unsigned 42000
Afiseaza 42000 ca int -23536
C:\>
```

Observație: *Atunci când compilați acest program cu* ***Turbo C++ Lite****, compilatorul va afișa două mesaje de eroare, deoarece compilatorul vede valoarea constantă 42000, pe care programul încearcă să o atribuie variabilei* ***unsigned int valoare****, ca număr de tip* ***long****, nu* ***int****. În acest caz, pentru că scopul programului este să arate erorile care pot apărea din declarațiile* ***unsigned int****, trebuie să ignorați avertismentele compilatorului. Alte compilatoare pe 16 biți vor afișa avertismente similare.*

57 AFIȘAREA VALORILOR DE TIP LONG INT FOLOSIND FUNCȚIA PRINTF



Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). Pentru a afișa valorile de tip *long int* cu funcția *printf*, trebuie să folosiți specificatorul de format *%ld*. Dacă folosiți *%d* în loc de *%ld*, *printf* va trata valoarea respectivă ca fiind de tip *int* și probabil că va afișa un rezultat greșit. Următorul program, *long_int.c*, folosește specificatorii de format *%ld* și *%d* pentru a afișa valoarea 1000000. Programul *long_int.c* ilustrează tipul de eroare care poate să apară dacă folosiți un specificator de format greșit:

```
#include <stdio.h>
void main(void)
  {
    long int un_milion = 1000000;

    printf("Un milion este %ld\n", un_milion);
    printf("Un milion is %d\n", un_milion);
  }
```

Atunci când veți compila și executa programul *long_int.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Un milion este 1000000
Un milion este 16960
C:\>
```

58 AFIȘAREA VALORILOR DE TIP FLOAT FOLOSIND FUNCȚIA PRINTF



Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). Pentru a afișa valorile de tip *float* cu funcția *printf*, trebuie să folosiți specificatorul de format *%f*. Următorul program, *floatout.c*, folosește specificatorul de format *%f* pentru a afișa valori în virgulă mobilă:

```
#include <stdio.h>
void main(void)
  {
    float pret = 525.75;
    float rata_impozit = 0.06;

    printf("Pretul este %f\n", pret);
    printf("Impozitul pe vanzari este %f\n",
      pret * rata_impozit);
  }
```

Atunci când veți compila și executa programul *long_int.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Pretul este 525.750000
Impozitul pe vanzari este 31.544999
C:\>
```

După cum puteți vedea, în mod prestabilit, specificatorul de format *%f* nu oferă decât o formatare simplă a ieșirii. Mai multe secțiuni din acest capitol vor prezenta însă diferite modalități de a formata ieșirea utilizând funcția *printf*.

AFIȘAREA VALORILOR DE TIP CHAR FOLOSIND FUNCȚIA PRINTF

C/C++ 59

Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). Pentru a afișa valorile de tip *char* cu funcția *printf*, trebuie să folosiți specificatorul de format *%c*. Următorul program, *char_out.c*, folosește specificatorul de format *%c* pentru a afișa pe ecran litera A:

```
#include <stdio.h>
void main(void)
   {
     printf("Litera este %c\n", 'A');
     printf("Litera este %c\n", 65);
   }
```

După cum vedeți, programul *char_out.c* va afișa litera A utilizând constanta de tip caracter 'A' și valoarea ASCII 65. Atunci când veți compila și executa programul *charout.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Litera este A
Litera este A
C:\>
```

AFIȘAREA VALORILOR ÎN VIRGULĂ MOBILĂ ÎN FORMAT EXPONENȚIAL

C/C++ 60

Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). În secțiunea 58, ați învățat că puteți afișa valori în virgulă mobilă prin utilizarea specificatorului de format *%f*. În funcție de cerințele programului, este posibil să afișați valori folosind formatul exponențial. Pentru a afișa valori în virgulă mobilă într-un format exponențial, folosiți specificatorul de format *%e* sau *%E*. Diferența dintre *%e și %E* este că specificatorul de format *%E* indică funcției *printf* să folosească majuscula E în ieșire. Următorul program, *expout.c*, folosește ambii specificatori de format exponențial pentru a afișa pe ecran valori în virgulă mobilă:

```
#include <stdio.h>
void main(void)
   {
     float pi = 3.14159;
     float raza = 2.0031;
```

```
    printf("Aria cercului este %e\n", 2 * pi * raza);
   printf("Aria cercului este %E\n", 2 * pi * raza);
 }
```

Atunci când veți compila și executa programul *expout.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Aria cercului este 1.258584e+01
Aria cercului este 1.258584E+01
C:\>
```

După cum vedeți, în mod prestabilit, specificatorii de format *%e* și *%E* nu oferă decât o formatare simplă a ieșirii. Mai multe secțiuni din acest capitol vor prezenta însă diferite modalități de a formata ieșirea utilizând funcția *printf*.

61 AFIȘAREA VALORILOR ÎN VIRGULĂ MOBILĂ



În secțiunea 58, ați învățat că, folosind specificatorul de format *%f*, puteți să indicați funcției *printf* să afișeze valori în virgulă mobilă în format zecimal. De asemenea, în secțiunea 60, ați învățat că puteți folosi specificatorii *%e* și *%E* pentru a afișa valori în virgulă mobilă în format exponențial. În mod similar, funcția *printf* acceptă specificatorii de format *%g* și *%G*. Atunci când folosiți specificatorii *%g* și *%G*, *printf* alege formatul *%f sau %e*, astfel încât să ofere utilizatorului o afișare cât mai convenabilă. Următorul program, *vrg_mob.c*, ilustrează utilizarea specificatorului de format *%g*:

```
#include <stdio.h>
void main(void)
  {
    printf("Numarul 0.1234 este afisat in formatul
      %g\n", 0.1234);
    printf("Numarul 0.00001234 este afisat in formatul %g\n",
      0.00001234);
  }
```

Atunci când compilați și executați programul *vrg_mob.c*, pe ecran va fi afișat următorul rezultat:

```
Numarul 0.1234 este afisat in formatul 0.1234
Numarul 0.00001234 este afisat in formatul 1.234e-05
C:\>
```

62 AFIȘAREA UNUI ȘIR DE CARACTERE FOLOSIND FUNCȚIA PRINTF



Un *șir de caractere* este o secvență care conține zero sau mai multe caractere. (Capitolul „Șirurile" vă oferă mai multe amănunte în legătură cu acest subiect.) Una dintre cele mai obișnuite operații executate în programe este afișarea șirurilor de caractere. Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int*, *float*, *char* și așa mai departe). Pentru a afișa un șir de caractere cu funcția *printf*, trebuie să folosiți specificatorul de format *%s*. Următorul program, *str_out.c*, utilizează specificatorului de format *%s* pentru a afișa un șir de caractere:

```
#include <stdio.h>
void main(void)
  {
    char titlu[255] = "Totul despre C/C++";
    printf("Numele acestei carti este %s\n", titlu);
  }
```

Atunci când compilați și executați programul *str_out.c*, pe ecran va fi afișat următorul rezultat:

```
Numele acestei carti este Totul despre C/C++
C:\>
```

AFIȘAREA POINTERILOR FOLOSIND FUNCȚIA PRINTF C/C++ 63

Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care furnizează informații despre tipurile parametrilor (cum ar fi *int, float, char* și așa mai departe). Ați învățat, de asemenea, că variabila este un nume pe care programul dumneavoastră o asociază unei locații de memorie. Pe măsură ce complexitatea programelor dumneavoastră va crește, veți lucra probabil cu adrese de memorie (numite *pointeri*). Când lucrați cu *pointeri*, poate apărea necesitatea de a afișa o adresă de memorie. Pentru a afișa această adreșă cu funcția *printf*, folosiți specificatorul de format *%p*. Următorul program, *ptr_out.c*, utilizează specificatorul de format *%p* pentru a afișa o adresă de memorie:

```
#include <stdio.h>
void main(void)
  {
    int valoare;
    printf("Adresa valorii variabilei este %p\n", &valoare);
  }
```

Atunci când compilați și executați programul *ptr_out.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Adresa valorii variabilei este FFF4
C:\>
```

Atunci când utilizați specificatorul de format *%p*, valoarea pointerului și formatul utilizat de *printf* diferă de la un sistem de operare la altul. Capitolul „Pointerii" va explica în detaliu modalitatea de utilizare a pointerilor.

UTILIZAREA SEMNULUI PLUS SAU MINUS ÎNAINTEA UNEI VALORI C/C++ 64

Așa cum ați învățat, funcția *printf* permite utilizarea unor specificatori de format care controlează modul de afișare a mesajelor. Când folosiți funcția *printf* pentru a afișa o valoare negativă, valoarea va fi întotdeauna precedată de semnul minus. În funcție de programul dumneavoastră, veți dori uneori ca *printf* să afișeze și semnul pentru valorile pozitive. Pentru a indica funcției *printf* să afișeze semnul valorii, pur și simplu veți include un semn plus

imediat după % în specificatorul de format. Următorul program, *aratsemn.c*, ilustrează utilizarea semnului plus pentru specificatorii de format:

```
#include <stdio.h>
void main(void)
  {
    int neg_int = -5;
    int poz_int = 5;
    float neg_flt = -100.23;
    float poz_flt = 100.23;
    printf("Valoarea intreaga este %+d si %+d\n", neg_int,
      poz_int);
    printf("Valoarea in virgula mobila este %+f %+f\n",
      neg_flt, poz_flt);
  }
```

Atunci când compilați și executați programul *aratsemn.c*, pe ecran va apărea următorul mesaj:

```
Valoarea intreaga este -5 si +5
Valoarea in virgula mobila este -100.230003 +100.230003
C:\>
```

65 FORMATAREA UNEI VALORI ÎNTREGI FOLOSIND FUNCȚIA PRINTF

C/C++

După cum ați citit în secțiunea 54, specificatorul de format *%d* indică funcției *printf* să afișeze o valoare întreagă. Pe măsură ce programul dumneavoastră va deveni mai complex, veți dori ca *printf* să formateze mai bine datele. De exemplu, să presupunem că doriți să afișați pe ecranul calculatorului un tabel similar cu următorul:

```
Vanzator   Cantitate
Ionescu    332
Popescu    1200
Georgescu  3311
Gheorghe   43
```

Dacă folosiți specificatorul de format *%d*, puteți să indicați funcției *printf* să afișeze un număr minim de caractere. Următorul program, *int_fmt.c*, ilustrează modul în care puteți formata valorile întregi folosind specificatorul *%d*:

```
#include <stdio.h>
void main(void)
  {
    int valoare = 5;

    printf ("%1d\n", valoare);
    printf ("%2d\n", valoare);
    printf ("%3d\n", valoare);
    printf ("%4d\n", valoare);
  }
```

Atunci când compilați și executați programul *int_fmt.c*, pe ecran vor apărea următoarele:

```
5
   5
      5
         5
C:\>
```

Cifra pe care o plasați după % precizează numărul minim de caractere pe care *printf* îl folosește pentru a afișa o valoare întreagă. Dacă, de exemplu, specificați *%5d* și valoarea pe care vreți să o afișați este 10, *printf* va introduce trei spații libere înainte de valoare. Rețineți că valoarea precizează numărul minim de caractere pe care îl va avea ieșirea. Dacă valoarea pe care doriți să o afișați are mai multe caractere decât ați precizat, *printf* va folosi numărul de caractere necesar pentru afișarea corectă a valorii.

AFIȘAREA NUMERELOR ÎNTREGI PRECEDATE DE ZERO C/C++ 66

În secțiunea 65, ați învățat cum să formatați o valoare întreagă plasând numărul de cifre dorit imediat după % în specificatorul de format *%d*. Dacă valoarea întreagă pe care o afișează *printf* nu are nevoie de numărul de caractere pe care le-ați specificat, va introduce un număr corespunzător de spații libere înaintea valorii. În funcție de scopul programului dumneavoastră, poate că veți dori ca *printf* să introducă zerouri înaintea valorii, în loc de spații. Pentru a indica funcției *printf* să adauge zerouri înaintea valorii, plasați un 0 imediat după % în specificatorul de format, înainte de numărul de cifre dorit. Următorul program, *zero_pad.c*, ilustrează acest procedeu:

```
#include <stdio.h>
void main(void)
  {
    int valoare = 5;

    printf ("%01d\n", valoare);
    printf ("%02d\n", valoare);
    printf ("%03d\n", valoare);
    printf ("%04d\n", valoare);
  }
```

Atunci când compilați și executați programul *zero_pad.c*, pe ecran vor apărea următoarele:

```
5
05
005
0005
C:\>
```

AFIȘAREA UNUI PREFIX ÎNAINTEA VALORILOR OCTALE ȘI HEXAZECIMALE C/C++ 67

În secțiunea 55, ați învățat cum se folosește specificatorul de format *%o* pentru a afișa valori octale și specificatorul de format *%x* și *%X* pentru a afișa valorile hexazecimale. Atunci când programul dumneavoastră are la ieșire astfel de valori, poate că veți dori ca valorile octale să fie precedate de zero (de exemplu *0777*), iar cele hexazecimale de *0x* (de exemplu *0xFF*).

Pentru a indica funcției *printf* să adauge prefixul potrivit unei valori octale sau hexazecimale, plasați caracterul # imediat după % în specificatorul de format. Următorul program, *oct_hex.c*, ilustrează modul de utilizare a caracterului # în specificatorii de format ai funcției *printf*:

```
#include <stdio.h>
void main(void)
  {
    int valoare = 255;
    printf("Valoarea zecimala %d este %#o in octal\n", valoare,
      valoare);
    printf("Valoarea zecimala %d este %#x in hexazecimal\n",
      valoare, valoare);
    printf("Valoarea zecimala %d este %#X in hexazecimal\n",
      valoare, valoare);
  }
```

Atunci când veți compila și executa programul *oct_hex.c*, pe ecran va apărea:

```
Valoarea zecimala 255 este 0377 in octal
Valoarea zecimala 255 este 0xff in hexazecimal
Valoarea zecimala 255 este 0xFF in hexazecimal
C:\>
```

68 FORMATAREA VALORILOR ÎN VIRGULĂ MOBILĂ



În secțiunea 65, ați învățat cum să formatați o valoare întreagă plasând numărul dorit de cifre imediat după % în specificatorul de format *%d*. Folosind o tehnică similară, funcția *printf* vă permite să formatați și valorile în virgulă mobilă. Atunci când formatați o valoare în virgulă mobilă, precizați două valori. Prima valoare indică funcției *printf* numărul minim de caractere pe care vreți să le afișați. A doua valoare comunică funcției *printf* numărul de cifre pe care doriți să le afișeze la dreapta punctului zecimal. Următorul program, *virg_fmt.c*, ilustrează modul de formatare a valorilor în virgulă mobilă folosind *printf*:

```
#include <stdio.h>

void main(void)
  {
    float valoare = 1.23456;

    printf("%8.1f\n", valoare);
    printf("%8.3f\n", valoare);
    printf("%8.5f\n", valoare);
  }
```

Atunci când veți compila și executa programul *virg_fmt.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
     1.2
   1.235
 1.23456
C:\>
```

FORMATAREA UNEI IEȘIRI EXPONENȚIALE

C/C++ 69

În secțiunea 68, ați învățat cum să folosiți specificatorul de format *%f* pentru a formata o valoare în virgulă mobilă. Folosind o tehnică similară de formatare, puteți cere funcției *printf* să afișeze valorile în virgulă mobilă, în format exponențial. Următorul program, *exp_fmt.c*, ilustrează formatarea unei ieșiri exponențiale:

```
#include <stdio.h>
void main(void)
  {
    float valoare = 1.23456;
    printf ("%12.1e\n", valoare);
    printf ("%12.3e\n", valoare);
    printf ("%12.5e\n", valoare);
  }
```

Atunci când veți compila și executa programul *exp_fmt.c*, ecranul dumneavoastră va afișa următoarea ieșire:

```
     1.2e+00
   1.235e+00
 1.23456e+00
C:\>
```

ALINIEREA LA STÂNGA A IEȘIRII

C/C++ 70

Atunci când afișați un text utilizând formatarea cu *printf*, alinierea prestabilită este la dreapta. În funcție de programul dumneavoastră, este posibil să doriți ca *printf* să alinieze rezultatul la stânga. Pentru a alinia la stânga un text, plasați un semn minus (–) imediat după % în specificatorul de format. Următorul program, *alin_stg.c*, ilustrează utilizarea semnului minus pentru a alinia la stânga un text:

```
#include <stdio.h>

void main(void)
  {
    int val_int = 5;
    float val_flt = 3.33;

    printf("Aliniere dreapta %5d a valorii\n", val_int);
    printf("Aliniere stanga %-5d a valorii\n", val_int);
    printf("Aliniere dreapta %7.2f a valorii\n", val_flt);
    printf("Aliniere stanga %-7.2f a valorii\n", val_flt);
  }
```

Atunci când veți compila și executa programul *alin_stg.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Aliniere dreapta     5 a valorii
Aliniere stanga 5      a valorii
Aliniere dreapta    3.33 a valorii
Aliniere stanga 3.33    a valorii
C:\>
```

71 UTILIZAREA COMBINATĂ A SPECIFICATORILOR DE FORMAT

C/C++

Multe dintre capitolele acestei secțiuni au prezentat diferiți specificatori de format. Vor apărea și situații în care veți dori să folosiți avantajele a doi sau mai mulți specificatori. De exemplu, puteți să doriți să afișați o valoare hexazecimală aliniată la stânga, precedată de caracterul *0x*. În acest caz, pur și simplu plasați fiecare specificator imediat după semnul *%*. Următorul program, *tot_fmt.c*, ilustrează utilizarea mai multor specificatori de format:

```
#include <stdio.h>

void main(void)
  {
    int val_int = 5;

    printf("Aliniere stanga cu semn %-+3d\n", val_int);
  }
```

Atunci când veți compila și veți executa programul *tot_fmt.c*, ecranul va afișa:

```
Aliniere stanga cu semn +5
C:\>
```

72 TRECEREA UNUI ȘIR DE CARACTERE PE LINIA URMĂTOARE

C/C++

Când programul dumneavoastră folosește funcția *printf*, poate apărea situația în care un șir de caractere nu încape pe un singur rând. În asemenea ocazii, pur și simplu plasați un caracter backslash (\) la sfârșitul liniei, ceea ce va face ca textul să continue pe linia următoare, cum se arată mai jos:

```
printf("Acest text este foarte lung si pentru ca este atat de \
lung, nu incape pe o singura linie.");
```

Observație: *Dacă treceți textul pe linia următoare, nu includeți spații la începutul liniei noi. Dacă există spații, compilatorul de C va include aceste spații în interiorul șirului de caractere.*

73 AFIȘAREA ȘIRURILOR NEAR ȘI FAR

C/C++

Capitolul „Memoria" explică în detaliu pointerii *near* și *far*. Pe scurt, pointerii *near* și *far* reprezintă adresele variabilelor în interiorul zonei de memorie alocate programului. Programele care rulează în cadrul sistemelor de operare mai vechi, cum ar fi MS-DOS, folosesc pointerii *far* pentru a mări zona de memorie în care programul poate să păstreze informația. Atunci când programul folosește pointeri *far* pentru referirea variabilelor de tip șir, poate apărea situția în care veți dori să afișați conținutul șirului folosind *printf*. Însă, așa cum veți învăța în capitolul „Funcțiile", compilatorul va genera o eroare dacă transmiteți un pointer *far* unei funcții care așteaptă o adresă *near*. Dacă doriți să afișați unui șir *far* (al cărui început este indicat de un pointer *far*) folosind funcția *printf*, trebuie să anunțați funcția că folosiți un pointer *far*. Pentru a face aceasta, plasați o majusculă F (pentru *far*) imediat după % în specificatorul de format, cum se arată în continuare:

```
printf("%Fs\n", un_sir_far);
```

Pentru că specificatorul *%Fs* arată funcției *printf* că utilizați un pointer *far*, apelarea funcției este corectă. În mod similar, puteți plasa o majusculă N în specificatorul de format pentru a indica funcției *printf* că-i transferați un șir *near*. Dar cum funcția *printf* așteptă în mod prestabilit șiruri *near*, specificatorii de format *%Ns* și *%s* dau același rezultat. Următorul program în C, *near_far.c*, ilustrează utilizarea specificatorilor *%Fs* și *%Ns* în cadrul funcției *printf*.

```
#include <stdio.h>

void main(void)
  {
    char *near_titlu = "Totul despre C/C++";
    char far *far_titlu = "Totul despre C/C++";

    printf("Titlul cartii este: %Ns\n", near_titlu);
    printf("Titlul cartii este: %Fs\n", far_titlu);
  }
```

Observație: ***Visual C++*** *nu face distincția între pointerii* ***far*** *și* ***near***. *Dacă încercați să compilați programul* ***near_far.c*** *sub* ***Visual C++,*** *compilatorul va returna o eroare. Pentru a actualiza automat programul dumneavoastră astfel încât să ruleze sub* ***Visual C++***, *includeți fișierul antet* ***windef.h*** *în program.*

CARACTERELE ESCAPE ALE FUNCȚIEI PRINTF

C/C++ 74

Când lucrați cu șiruri de caractere, puteți să utilizați caractere speciale, cum ar fi tabulatoarele, retur de car sau avans de rând. C definește mai multe *caractere escape* (caractere precedate de simbolul escape, backslash) pentru a facilita includerea caracterelor speciale în interiorul șirurilor. De exemplu, multe dintre programele prezentate în această carte au folosit caracterul linie nouă (\n) pentru ca textul de la ieșire să înceapă pe linia următoare, cum se poate vedea mai jos:

```
printf("Linie 1\nLinie 2\nLinie 3\n");
```

Tabelul 74 prezintă lista caracterelor escape pe care le puteți utiliza în cadrul șirurilor de caractere (și deci în ieșirile create cu funcția *printf*).

Caracter escape	Semnificație
\a	Caracter ASCII de atenționare
\b	Backspace
\f	Avans hârtie
\n	Linie nouă
\r	Retur de car
\t	Tabulator orizontal
\v	Tabulator vertical

(continuare)

Caracter escape	Semnificație
\\	Backslash
\'	Apostrof
\"	Ghilimele
\?	Semnul întrebării
nnn	Valoare ASCII în octal
xnnn	Valoare ASCII în hexazecimal

***Tabelul 74** Caracterele escape definite în C.*

75 DETERMINAREA NUMĂRULUI DE CARACTERE AFIȘATE DE PRINTF

C/C++

Când programul dumneavoastră execută formatări sofisticate, apar situații în care veți dori să știți numărul de caractere pe care le-a afișat funcția *printf.* Atunci când folosiți specificatorul de format *%n, printf* va atribui unei variabile (în urma unui transfer prin pointer) suma totală a caracterelor pe care le-a afișat. Următorul program, *nr_crt.c,* ilustrează utilizarea specificatorului de format *%n:*

```
#include <stdio.h>

void main(void)
  {
    int prima_suma;
    int a_doua_suma;

    printf("Totul despre C/C++%n\n", &prima_suma,
      &a_doua_suma);
    printf("Prima suma %d A doua suma %d\n", prima_suma,
      a_doua_suma);
  }
```

Atunci când veți compila și executa programul *nr_crt.c,* ecranul va afișa următoarele:

```
Totul despre C/C++
Prima suma 5 A doua suma 35
C:\>
```

76 UTILIZAREA VALORII RETURNATE

C/C++

În secțiunea 75, ați învățat cum se folosește specificatorul de format *%n* pentru a determina numărul de caractere pe care le-a scris *printf.* Folosirea specificatorului de format *%n* este una dintre modalitățile prin care vă puteți asigura că funcția *printf* a afișat corect rezultatul. În plus, când *printf* a terminat execuția, returnează numărul total de caractere pe care le-a scris. Dacă *printf* întâlnește o eroare, va returna constanta EOF (care, așa cum veți învăța, indică sfârșitul fișierului). Următorul program, *printfok.c,* folosește valoarea returnată pentru a se asigura că funcția *printf* s-a executat cu succes:

```
#include <stdio.h>
void main(void)
  {
    int rezultat;

    rezultat = printf("Totul despre C/C++\n");
    if (rezultat == EOF)
      fprintf(stderr, "Eroare la printf\n");
  }
```

Dacă utilizatorul a redirectat ieșirea către un fișier sau periferic (cum ar fi imprimanta) și dacă redirectarea I/O întâmpină o eroare (cum ar fi *device off-line* sau *disk full*), programul dumneavoastră poate să detecteze eroarea prin testarea valorii returnate de *printf*.

UTILIZAREA DRIVERULUI DE DISPOZITIV ANSI — C/C++ 77

Multe dintre secțiunile prezentate de-a lungul acestei cărți s-au referit pe larg la capacitatea funcției *printf* de a formata rezultatul la afișare. În afara specificatorilor care vă permit să controlați numărul de cifre afișate, să afișați rezultatul în octal sau hexazecimal sau să aliniați textul la stânga sau la dreapta, funcția *printf* nu furnizează alți specificatori de format. *Printf* nu vă oferă specificatori de format care să vă aducă cursorul într-un anumit rând sau coloană, să șteargă ecranul sau să afișeze în culori. Însă, în funcție de sistemul de operare pe care îl folosiți, puteți să realizați aceste operații folosind driverul de dispozitiv ANSI. Driverul ANSI acceptă diferite secvențe escape care îi indică să folosească anumite culori, să poziționeze cursorul sau chiar să elibereze ecranul. Aceste instrucțiuni de formatare sunt numite de programatori *secvențe escape* pentru că ele încep cu caracterul ASCII escape (valoarea 27). Dacă folosiți sistemul de operare DOS, instalați driverul ANSI prin plasarea unei intrări în fișierul *config.sys* (vezi exemplul de mai jos), apoi reîncărcați sistemul.

```
DEVICE=C:\DOS\ANSI.SYS
```

După ce ați instalat driverul ANSI, programul poate scrie secvențe escape folosind *printf*.

Observație: *Dacă pe calculatorul cu care vă compilați programele rulați și Windows 95, adăugarea driverului ANSI în fișierul* ***config.sys*** *nu va interfera cu operațiile din Windows 95.*

UTILIZAREA DRIVERULUI ANSI PENTRU A ELIBERA ECRANUL — C/C++ 78

Una dintre cele mai obișnuite operații pe care le va efectua un program la începutul execuției sale, este să elibereze ecranul. Din păcate, biblioteca run-time C nu conține o funcție care să elibereze ecranul. Pentru a face aceasta, utilizați driverul ANSI, cum s-a prezentat în secțiunea 77, și apoi invocați următoarea secvență escape:

```
Esc[2j
```

O modalitate ușoară de a invoca o secvență escape este reprezentarea în octal a caracterului escape (\033). Tipăriți caracterul escape în felul următor:

```
printf("\033[2J");
```

79 UTILIZAREA DRIVERULUI ANSI PENTRU A AFIȘA CULORI PE ECRAN



În multe dintre capitolele acestei cărți a fost folosită frecvent funcția *printf* pentru a afișa o ieșire. Deși funcția *printf* oferă specificatori de format puternici, nu vă oferă și mijloacele de a afișa ieșirea în culori. Dacă însă folosiți driverul ANSI, prezentat în secțiunea 77, puteți folosi secvențele escape prezentate în tabelul 79 pentru a afișa ieșirea în culori.

Secvență escape	Culoare
Esc[30m	Negru – culoare de prim-plan
Esc[31m	Roșu – culoare de prim-plan
Esc[32m	Verde – culoare de prim-plan
Esc[33m	Portocaliu – culoare de prim-plan
Esc[34m	Albastru – culoare de prim-plan
Esc[35m	Magenta – culoare de prim-plan
Esc[36m	Cian – culoare de prim-plan
Esc[37m	Alb – culoare de prim-plan
Esc[40m	Negru – culoare de fundal
Esc[41m	Roșu – culoare de fundal
Esc[42m	Verde – culoare de fundal
Esc[43m	Portocaliu – culoare de fundal
Esc[44m	Albastru – culoare de fundal
Esc[45m	Magenta – culoare de fundal
Esc[46m	Cian – culoare de fundal
Esc[47m	Alb – culoare de fundal

***Tabelul 79** Secvențele escape ANSI pentru stabilirea culorilor de ecran.*

Următoarea instrucțiune *printf* selectează culoarea de fond albastru:

```
printf("\033[44m");
```

În mod similar, următoarea instrucțiune *printf* selectează un text roșu pe un fundal alb:

```
printf("\033[47m\033[31m");
```

În exemplul precedent, *printf* scrie două secvențe escape. Driverul ANSI permite specificarea culorilor de ecran prin separarea cu punct și virgulă (;), ca mai jos:

```
printf("\033[47m;31m");
```

UTILIZAREA DRIVERULUI ANSI PENTRU POZIŢIONAREA CURSORULUI

C/C++ 80

Aşa cum aţi învăţat, driverul ANSI acceptă secvenţe escape care permit ştergerea ecranului şi afişarea textului în culori. În plus, driverul ANSI dispune de secvenţe escape care permit plasarea cursorului la o poziţie specifică pe rând şi coloană, astfel că puteţi afişa ieşirea într-o anumită locaţie de pe ecran. Tabelul 80 arată secvenţele escape ale driverului ANSI pentru poziţionarea cursorului:

Secvenţă escape	Funcţie	Exemplu
Esc[x;yH	Poziţionează cursorul la rândul x şi coloana y	*Esc[10;25H*
Esc[xA	Mută cursorul cu x rânduri mai sus	*Esc[1A*
Esc[xB	Mută cursorul cu x rânduri mai jos	*Esc[2B*
Esc[yC	Mută cursorul y coloane la dreapta	*Esc[10C*
Esc[yD	Mută cursorul y coloane la stânga	*Esc[10D*
Esc[S	Stochează poziţia curentă a cursorului	*Esc[S*
Esc[U	Restaurează poziţia curentă a cursorului	*Esc[U*
Esc[2j	Şterge ecranul şi mută cursorul la poziţia home	*Esc[2j*
Esc[K	Şterge până la sfârşitul liniei curente	*Esc[K*

Tabelul 80 *Secvenţele escape ale driverului ANSI pentru poziţionarea cursorului.*

REALIZAREA OPERAŢIILOR MATEMATICE DE BAZĂ ÎN C

C/C++ 81

Chiar şi în cele mai simple programe, veţi efectua operaţii aritmetice cum ar fi adunarea, scăderea, înmulţirea sau împărţirea. Pentru a realiza aceste operaţii matematice de bază, folosiţi operatorii descrişi în tabelul 81.

Operator	Funcţie
+	Adunare
–	Scădere
*	Înmulţire
/	Împărţire

Tabelul 81 *Operaţiile matematice de bază în C.*

Următorul program, *math.c*, ilustrează utilizarea operatorilor de bază din C:

```
#include <stdio.h>

void main(void)
   {
     int secunde_pe_ora;
     float media;

     secunde_pe_ora = 60 * 60;
     media = (5 + 10 + 15 + 20) / 4;
```

```
    printf("Numarul de secunde intr-o ora este %d\n",
      secunde_pe_ora);
    printf("Media numerelor 5, 10, 15 si 20 este %f\n", media);
    printf("Numarul de secunde in 48 minute este %d\n",
      secunde_pe_ora - 12 * 60);
  }
```

Atunci când compilaţi şi executaţi programul *math.c*, pe ecranul dumnevoastră va fi afişată următoarea ieşire:

```
Numarul de secunde intr-o ora este 3600
Media numerelor 5, 10, 15 si 20 este 12.000000
Numarul de secunde in 48 minute este 2880
C:\>
```

82 OPERATORUL MODULO C/C++

În secţiunea 81, aţi învăţat că C foloseşte caracterul slash (/) ca operator pentru împărţire. În unele aplicaţii, probabil că veţi avea nevoie de restul unei împărţiri de numere întregi. În asemenea cazuri, folosiţi opratorul *modulo* (rest). Următorul program, *modulo.c*, ilustrează modul în care se utilizează acest operator:

```
#include <stdio.h>

void main(void)
  {
    int rest;
    int rezultat;

    rezultat = 10 / 3;
    rest = 10 % 3;
    printf("10 impartit la 3 egal %d rest %d\n", rezultat, rest);
  }
```

Atunci când compilaţi şi executaţi programul *modulo.c*, pe ecranul dumnevoastră va fi afişată următoarea ieşire:

```
10 impartit la 3 egal 3 rest 1
C:\>
```

83 PRECEDENŢA ŞI ASOCIATIVITATEA OPERATORILOR C/C++

În secţiunea 81, aţi învăţat că limbajul C foloseşte următorii operatori: semnul plus (+) pentru adunare, semnul minus (–) pentru scădere, asteriscul (*) pentru înmulţire şi caracterul slash (/) pentru împărţire. Atunci când programul dumneavoastră foloseşte aceşti operatori în interiorul unor expresii aritmetice, este necesar să înţelegeţi precedenţa operatorilor în C, care specifică ordinea în care se efectuează operaţiile aritmetice. De exemplu, să considerăm următoarea expresie:

```
rezultat = 5 + 2 * 3;
```

Dacă limbajul C ar efectua operațiile de la stânga la dreapta (adunarea înaintea înmulțirii), rezultatul expresiei ar fi 21:

```
rezultat = 5 + 2 * 3;
         = 7 * 3;
         = 21;
```

Dacă C ar efectua mai întâi înmulțirea, rezultatul ar fi 11:

```
rezultat = 5 + 2 * 3;
         = 5 + 6;
         = 11;
```

Pentru a evita problemele cauzate de rezultatele nedeterminate, în limbajul C este definită *precedența operatorilor*. Precedența operatorilor determină care operații se vor efectua mai întâi. Tabelul 83 ilustrează precedența operatorilor în C:

Precedența operatorilor (în ordine descrescătoare)

<table>
<tr><td>()</td><td>[]</td><td>.</td><td>-></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>++</td><td>--</td><td>+</td><td>–</td><td>*</td><td>&</td><td>!</td><td>~</td><td>sizeof</td><td></td><td></td></tr>
<tr><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>+</td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>>></td><td><<</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>==</td><td>!=</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>^</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>|</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>&&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>?:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>=</td><td>+=</td><td>–=</td><td>*=</td><td>/=</td><td>%=</td><td>&=</td><td>^=</td><td>|=</td><td><<=</td><td>>>=</td></tr>
</table>

Tabelul 83 *Precedența operatorilor în C*

Atunci când creați o expresie, limbajul C va efectua mai întâi operațiile cu precedența cea mai mare. Dacă doi operatori au aceeași precedență, limbajul C va efectua operațiile de la stânga la dreapta.

FORȚAREA ORDINII OPERAȚIILOR

C/C++ 84

Așa cum ați citit în secțiunea 83, C efectuează operațiile dintr-o expresie bazându-se pe precedența fiecărui operator în interiorul expresiei. În multe cazuri, ordinea pe care o va stabili limbajul C nu este cea pe care o doriți. De exemplu, să considerăm următoarea expresie – scopul ei fiind calcularea mediei a trei valori:

```
media = 5 + 10 + 15 / 3;
```

Dacă media valorilor 5, 10 şi 15 ar fi calculată corect, rezultatul ar trebui să fie 10. Dar dacă lăsaţi limbajul C să evalueze expresia precedentă, el va atribui variabilei *media* valoarea 20, ca mai jos:

```
media  = 5 + 10 + 15 / 3;
       = 5 + 10 + 5;
       = 15 + 5;
       = 20;
```

Dacă examinaţi tabelul de precedenţă a operatorilor, pe care îl prezintă secţiunea 83, veţi vedea că operatorul de *împărţire* al limbajului C (/) are o precedenţă mai mare decât a operatorului de *adunare* (+). Prin urmare, aveţi nevoie de o modalitate pentru a schimba ordinea în care C efectuează operaţiile. Atunci când C evaluează o expresie, el va efectua întotdeauna operaţiile care apar în interiorul parantezelor înaintea altor operaţii. Dacă grupaţi în paranteză valorile pe care doriţi să le adunaţi, el va calcula media corectă, ca mai jos:

```
media  = (5 + 10 + 15) / 3;
       = (15 + 15) / 3;
       = (30) / 3;
       = 10;
```

Limbajul C execută operaţiile din interiorul parantezelor bazându-se pe regulile de precedenţă a operatorilor. Dacă o expresie conţine mai multe expresii cu mai multe seturi de paranteze, C execută mai întâi operaţiile din parantezele care nu includ alte paranteze, ca în exemplul de mai jos:

```
rezultat = ((5 + 3) * 2) - 3;
         = ((8) * 2) - 3;
         = (16) - 3;
         = 13;
```

85 OPERATORUL DE INCREMENTARE



Una dintre operaţiile efectuate cel mai frecvent în programe este incrementarea valorii curente a unei variabile cu 1. De exemplu, următoare instrucţiune creşte valoarea variabilei *variabila*:

```
variabila = variabila + 1;
```

Deoarece operaţiile de incrementare sunt atât de frecvente, limbajul C oferă o notaţie prescurtată pentru incrementarea variabilelor din programe, denumită *operator de incrementare*. Următoarea instrucţiune foloseşte *operatorul de incrementare* pentru a adăuga 1 la valoarea *variabilei*:

```
variabila++;
```

Următorul program, *0_la_100.c*, utilizează operatorul de incrementare pentru a tipări valorile de la 0 la 100:

```
#include <stdio.h>
void main(void)
  {
    int val = 0;
    while (val <= 100)
      {
        printf("%d\n", val);
        val++;
      }
  }
```

Limbajul C prevede două forme pentru operatorul de incrementare: forma *prefix* și forma *postfix*. Ambele instrucțiuni care urmează incrementează variabila *total* cu 1:

```
total++;
++total;
```

Prima instrucțiune folosește operatorul de *incrementare postfix*, iar a doua instrucțiune, operatorul de *incrementare prefix*. Nu trebuie să confundați cei doi operatori, deoarece limbajul C îi tratează în mod diferit. Atunci când utilizați operatorul de incrementare postfix, limbajul C va utiliza mai întâi valoarea variabilei și apoi va efectua operația de incrementare. În celălalt caz, când utilizați operatorul de incrementare prefix, limbajul C va incrementa mai întâi valoarea variabilei și apoi va utiliza variabila. Pentru a înțelege mai bine diferența dintre operatorul de incrementare prefix și cel postfix, observați cu atenție următorul program, *pre_post.c*, care utilizează ambii operatori:

```
#include <stdio.h>
void main(void)
  {
    int val = 1;

    printf("Utilizarea postfix %d\n", val++);
    printf("Valoarea dupa incrementare %d\n", val);
    val = 1;
    printf("Utilizarea prefix %d\n", ++val);
    printf("Valoarea dupa incrementare %d\n", val);
  }
```

Atunci când compilați și executați programul *pre_post.c*, pe ecranul dumnevoastră va fi afișată următoarea ieșire:

```
Utilizarea postfix 1
Valoarea dupa incrementare 2
Utilizarea prefix 2
Valoarea dupa incrementare 2
C:\>
```

După cum vedeți, când se utilizează *operatorul postfix*, limbajul C utilizează mai întâi valoarea variabilei, afișând valoarea 1, apoi incrementează variabila, obținând 2. Atunci când utilizați *operatorul prefix*, limbajul C incrementează mai întâi variabila, obținând 2, apoi afișează valoarea incrementată deja.

86 OPERATORUL DE DECREMENTARE



Aşa cum de multe ori veţi dori să incrementaţi valoarea unei variabile, la fel de frecvent veţi dori să decrementaţi cu 1 valoarea curentă a unei variabile, ca mai jos:

```
variabila = variabila - 1;
```

Deoarece operaţiile de decrementare sunt atât de frecvente, limbajul C oferă o notaţie prescurtată pentru aceste operaţii, denumită *operator de decrementare*. Următoarea instrucţiune foloseşte operatorul de decrementare pentru a scădea 1 din valoarea variabilei:

```
variabila--;
```

Ca şi în cazul operatorului de incrementare, limbajul C prevede două forme pentru operatorul de decrementare: forma *prefix* şi forma *postfix*. Ambele instrucţiuni care urmează decrementează variabila *total* cu 1:

```
total--;
--total;
```

Prima instrucţiune foloseşte operatorul de *decrementare postfix*, iar a doua foloseşte operatorul de *decrementare prefix*. Nu trebuie să confundaţi cei doi operatori, deoarece limbajul C îi tratează în mod diferit. Atunci când utilizaţi operatorul de decremenrtare postfix, limbajul C va utiliza mai întâi valoarea variabilei şi apoi va efectua operaţia de decrementare. În al doilea caz, când utilizaţi operatorul de decrementare prefix, limbajul C va decrementa mai întâi valoarea variabilei şi apoi va utiliza variabila. Pentru a înţelege mai bine diferenţa dintre operatorul de decrementare prefix şi cel postfix, studiaţi următorul program, *post_pre.c*, care utilizează ambii operatori:

```
#include <stdio.h>

void main(void)
  {
    int val = 1;

    printf("Utilizarea postfix %d\n", val--);
    printf("Valoarea dupa decrementare %d\n", val);
    val = 1;
    printf("Utilizarea prefix %d\n", --val);
    printf("Valoarea dupa decrementare %d\n", val);
  }
```

Atunci când compilaţi şi executaţi programul *post_pre.c*, pe ecranul dumnevoastră se va afişa următorul rezultat:

```
Utilizarea postfix 1
Valoarea dupa decrementare 0
Utilizarea prefix 0
Valoarea dupa decrementare 0
C:\>
```

După cum vedeți, când se utilizează *operatorul postfix*, limbajul C utilizează mai întâi valoarea variabilei, afișând valoarea 1, apoi decrementează variabila, obținând 0. Atunci când utilizați *operatorul prefix*, limbajul C decrementează mai întâi variabila, obținând 0, apoi afișează valoarea decrementată deja.

OPERAȚIA SAU PE BIȚI

C/C++ 87

Pe măsură ce complexitatea programelor dumneavoastră va crește, veți observa că puteți să măriți performanțele unui program sau să-i reduceți consumul de memorie utilizând *operații pe biți*. Operațiile pe biți sunt operații care se aplică unuia sau mai multor biți dintr-o valoare. Atunci când trebuie să manipulați o valoare bit cu bit, puteți folosi avantajele *operatorului SAU pe biți* (|) al limbajului C. Operatorul *SAU pe biți* examinează fiecare bit din două valori și obține o a treia valoare ca rezultat. De exemplu, să presupunem că două variabile au valorile 3 și 4, ai căror biți sunt respectiv 00000011 și 00000100. Operatorul *SAU pe biți* întoarce valoarea 7, așa cum arătăm mai jos:

```
3   00000011
4   00000100
    --------
7   00000111
```

În valoarea 3, biții 0 și 1 au valoarea unu, iar toți ceilalți biți au valoarea zero. În valoarea 4, bitul 2 are valoarea unu, iar ceilalți, zero. Rezultatul aplicării operației *SAU pe biți* va avea valoarea unu de fiecare dată când apare un 1 într-una dintre cele două valori inițiale. În acest caz, rezultatul are valoarea unu pentru biții 0, 1 și 2. Următorul program, *bit_sau.c*, ilustrează modul de utilizare a operatorului *SAU pe biți*:

```
#include <stdio.h>

void main(void)
  {
    printf("0 | 0 este %d\n", 0 | 0);
    printf("0 | 1 este %d\n", 0 | 1);
    printf("1 | 1 este %d\n", 1 | 1);
    printf("1 | 2 este %d\n", 1 | 2);
    printf("128 | 127 este %d\n", 128 | 127);
  }
```

Atunci când compilați și executați programul *bit_sau.c*, pe ecranul monitorului dumneavoastră se va afișa:

```
0 | 0 este 0
0 | 1 este 1
1 | 1 este 1
1 | 2 este 3
128 | 127 este 255
C:\>
```

88 OPERAȚIA ȘI PE BIȚI



Așa cum am mai spus în secțiunea 87, veți vedea că puteți mări performanțele unui program sau să-i reduceți consumul de memorie utilizând *operații pe biți*. Operațiile pe biți sunt operații care se aplică unuia sau mai multor biți dintr-o valoare. Atunci când trebuie să manipulați o valoare bit cu bit, puteți folosi avantajele *operatorului ȘI pe biți* al limbajului C (&). Operatorul *ȘI pe biți* examinează fiecare bit din două valori și obține o a treia valoare ca rezultat. De exemplu, să presupunem că două variabile au valorile 5 și 7, ai căror biți sunt respectiv 00000101 și 00000111. Operatorul *ȘI pe biți* returnează valoarea 5, așa cum se vede mai jos:

```
5    00000101
7    00000111
     --------
5    00000101
```

Dacă ambii biți din cei doi termeni au valoarea 1, atunci aplicarea operatorului *ȘI pe biți* va atribui valoarea 1 bitului corespunzător al rezultatului. Dacă cel puțin unul dintre cei doi termeni conține valoarea 0, atunci operatorul pe biți *ȘI* va atribui valoarea 0 bitului corespunzător al rezultatului. În cazul nostru, biții 0 și 2 au valoarea 1 în ambii termeni, deci rezultatul va fi 1 pentru acești biți și 0 pentru ceilalți biți. Următorul program, *bit_si.c*, ilustrează modul de utilizare a operatorului *ȘI pe biți*:

```
#include <stdio.h>

void main(void)
  {
    printf("0 & 0 este %d\n", 0 & 0);
    printf("0 & 1 este %d\n", 0 & 1);
    printf("1 & 1 este %d\n", 1 & 1);
    printf("1 & 2 este %d\n", 1 & 2);
    printf("15 & 127 este %d\n", 15 & 127);
  }
```

Atunci când compilați și executați programul *bit_si.c*, pe ecranul monitorului dumneavoastră se va afișa următorul rezultat:

```
0 & 0 este 0
0 & 1 este 0
1 & 1 este 1
1 & 2 este 0
15 & 127 este 15
C:\>
```

89 OPERAȚIA SAU EXCLUSIV PE BIȚI



După cum ați învățat deja, operațiile pe biți sunt operații care se aplică unuia sau mai multor biți dintr-o valoare. Atunci când trebuie să manipulați o valoare bit cu bit, pot apărea situații în care trebuie să apelați la avantajele operatorului *SAU exclusiv pe biți* (^), care analizează biții a două valori și stabilește biții rezultatului conform tabelei de adevăr prezentate în tabelul 89.

X	Y	Rezultat
0	0	0
0	1	1
1	0	1
1	1	0

Tabelul 89 *Bitul rezultat prin operația* ***SAU exclusiv pe biți.***

Să presupunem că două variabile au valorile 5 și 7, ai căror biți sunt 00000101 și respectiv 00000111. Operația *SAU exclusiv pe biți* returnează valoarea 2, așa cum se vede mai jos:

```
5    00000101
7    00000111
     --------
2    00000010
```

Următorul program, *bit_saux.c*, ilustrează utilizarea operatorului *SAU exclusiv pe biți*:

```
#include <stdio.h>

void main(void)
  {
    printf("0 ^ 0 este %d\n", 0 ^ 0);
    printf("0 ^ 1 este %d\n", 0 ^ 1);
    printf("1 ^ 1 este %d\n", 1 ^ 1);
    printf("1 ^ 2 este %d\n", 1 ^ 2);
    printf("15 ^ 127 este %d\n", 15 ^ 127);
  }
```

Atunci când compilați și executați programul *bit_saux.c*, pe ecranul dumneavoastră se va afișa următorul rezultat:

```
0 ^ 0 este 0
0 ^ 1 este 1
1 ^ 1 este 0
1 ^ 2 este 3
15 ^ 127 este 112
C:\>
```

OPERAȚIA DE COMPLEMENTARE PE BIȚI

C/C++ 90

După cum ați învățat, operațiile pe biți sunt operații care se aplică unuia sau mai multor biți dintr-o valoare. Atunci când trebuie să manipulați o valoare bit cu bit, puteți să apelați la avantajele operatorului *complementare pe biți* (~) al limbajului C. Operatorul *complementare pe biți* analizează fiecare bit dintr-o valoare și produce o a doua valoare ca rezultat. Operația *complementare pe biți* atribuie valoarea zero biților care conțin unu în valoarea inițială și valoarea unu celor care conțin zero. De exemplu, să presupunem că o variabilă de tip *unsigned* are valoarea 15. Operația *complementare pe biți* va returna 240, așa cum se arată mai jos:

```
15    00001111
240   11110000
```

După cum vedeți, biții care conțineau unu în valoarea inițială devin zero, iar cei care conțineau zero devin unu în valoarea rezultată. Următorul program, *bit_inv.c*, ilustrează modul de utilizare a operatorului *complementare pe biți*:

```
#include <stdio.h>

void main(void)
   {
     int val = 0xFF;

     printf("%X complementat este %X\n", val, ~val);
   }
```

Atunci când compilați și executați programul *bit_inv.c*, pe ecranul dumneavoastră se va afișa următoarea ieșire:

```
FF complementat este FF00
C:\>
```

91 APLICAREA UNOR OPERAȚII VALORILOR DE VARIABILE



Atunci când efectuați operații aritmetice într-un program, veți vedea că adesea atribuiți unei variabile rezultatul unei expresii care include valoarea curentă a variabilei respective, așa cum se poate vedea în instrucțiunile următoare:

```
total = total + 100;
suma = suma - 5;
jumatate = jumatate / 2;
```

Pentru cazurile în care operația de atribuire actualizează o variabilă cu rezultatul unei operații asupra valorii curente a variabilei, limbajul C dispune de o notație prescurtată pentru reprezentarea operației. Pe scurt, se plasează operatorul în fața operatorului de atribuire. Următoarele instrucțiuni sunt echivalente cu cele trei de mai sus:

```
total += 100;
suma -= 5;
jumatate /= 2;
```

Mai jos, puteți vedea alte exemple de instrucțiuni care sunt echivalente:

```
variabila += 10;         variabila = variabila + 10;
variabila <<= 2;         variabila = variabila << 2;
variabila &= 0xFF;       variabila = variabila & 0xFF;
variabila *= 1.05;       variabila = variabila * 1.05;
```

92 OPERATORUL CONDIȚIONAL AL LIMBAJULUI C



După cum veți învăța, instrucțiunea *if-else* din C examinează o condiție și execută un set de operații dacă este adevărată și un alt set de operații dacă este falsă. În mod similar, limbajul C pune la dispoziție *operatorul condițional*, care examinează o condiție și returnează o valoare dacă este adevărată și alta dacă este falsă. Formatul *operatorului condițional* este următorul:

```
(conditie) ? rezultatAdevarat : rezultatFals
```

Pentru a înțelege mai bine *operatorul condițional*, să considerăm următoarea condiție, care testează dacă un punctaj este mai mare sau egal cu 60. Dacă valoarea este mai mare sau egală cu 60, instrucțiunea atribuie variabilei *calificativ* valoarea A, adică „Admis". Dacă valoarea este mai mică decât 60, instrucțiunea atribuie variabilei *calificativ* valoarea R, pentru „Respins".

```
calificativ = (punctaj >= 60) ? 'A' : 'R'
```

Instrucțiunea este similară cu următoarea instrucțiune *if-else*:

```
if(punctaj >= 60)
   calificativ = 'A';
else
   calificativ = 'R';
```

Următoarea instrucțiune *printf* afișează șirul de caractere „Admis" sau „Respins" pe baza testării punctajului:

```
printf("Punctaj %d Rezultat %S\n", punctaj,
   (punctaj >= 60) ? 'Admis' : 'Respins');
```

Atunci când folosiți operatorul condițional pentru a atribui un rezultat condițional unei variabile, puteți reduce numărul instrucțiunilor *if-else* din cadrul programului.

OPERATORUL SIZEOF AL LIMBAJULUI C

C/C++ 93

Când programul declară o variabilă, compilatorul de C alocă memorie pentru stocarea valorii variabilei. Când scrieți programe care execută operații de intrare-ieșire cu fișiere sau alocă memorie pentru liste dinamice, veți găsi convenabil să aflați cantitatea de memorie alocată de program pentru o anumită variabilă. Operatorul C *sizeof* returnează numărul de biți alocați unei variabile sau unui tip. Următorul program, *sizeof.c*, exemplifică folosirea operatorului *sizeof*:

```
#include <stdio.h>

void main(void)
  {
    printf("Variabila de tip int foloseste %d octeti\n",
      sizeof(int));
    printf("Variabila de tip float foloseste %d octeti\n",
     sizeof(float));
    printf("Variabila de tip double foloseste %d octeti\n",
     sizeof(double));
    printf("Variabila de tip unsigned foloseste %d octeti\n",
     sizeof(unsigned));
    printf("Variabila de tip long foloseste %d octeti\n",
     sizeof(long));
  }
```

În funcție de compilatorul utilizat și de sistemul hardware, ieșirea rezultată la executarea programului *sizeof.c* poate fi diferită. Dacă folosiți compilatorul *Turbo C++ Lite*, va fi afișat următorul rezultat:

```
Variabila de tip int foloseste 2 octeti
Variabila de tip float foloseste 4 octeti
Variabila de tip double foloseste 8 octeti
Variabila de tip unsigned foloseste 2 octeti
Variabila de tip long foloseste 4 octeti
C:\>
```

94 EXECUTAREA DEPLASĂRII PE BIȚI



Când lucrați cu valori la nivel de bit, veți executa de obicei deplasări pe biți, fie la dreapta (dinspre cel mai semnificativ bit), fie la stânga (spre cel mai semnificativ bit). Pentru a permite executarea deplasărilor pe biți, limbajul C oferă doi operatori de deplasare pe biți: un operator care deplasează biții la dreapta (>>) și unul care deplasează biții la stânga (<<). Următoarea instrucțiune folosește operatorul de deplasare la stânga pentru a deplasa valorile variabilei *flag* două poziții spre stânga:

```
flag = flag << 2;
```

Să presupunem că variabila *flag* are valoarea 2, ca mai jos:

```
0000 0010
```

Când deplasați valoarea două locuri spre stânga, rezultatul va fi 8, așa cum se vede mai jos:

```
0000 1000
```

Când deplasați valorile spre stânga, compilatorul C scrie zerouri în poziția bitului celui mai puțin semnificativ. În schimb, când deplasați valorile spre dreapta, valoarea pe care C o pune în poziția bitului cel mai semnificativ depinde de tipul variabilei. Dacă variabila este de tip *unsigned*, compilatorul C introduce zerouri în bitul cel mai semnificativ în timpul operației de deplasare la dreapta. Însă, dacă variabila este de tip *signed* (cu alte cuvinte, dacă nu ați declarat-o ca variabilă *unsigned*), C folosește valoarea 1 dacă valoarea este negativă sau 0 dacă valoarea este pozitivă. Următorul program, *dpls.c*, ilustrează modul de utilizare al operațiilor de deplasare la dreapta și la stânga pe biți:

```
#include <stdio.h>
void main(void)
  {
    unsigned u_val = 1;
    signed int val = -1;
    printf("%u (unsigned) deplasat la stanga de 2 ori este
     %u\n", u_val, u_val << 2);
    printf("%u (unsigned) deplasat la dreapta de 2 ori este %u\n",
    u_val, u_val >> 2);
    u_val = 65535;
    printf("%u (unsigned) deplasat la stanga de 2 ori este
      %u\n", u_val, u_val << 2);
```

```
    printf("%u (unsigned) deplasat la dreapta de 2 ori este %u\n",
      u_val, u_val >> 2);
    printf("%d (signed) deplasat la stanga de 2 ori este %d\n",
      val, val << 2);
    printf("%d (signed) deplasat la dreapta de 2 ori este %d\n",
      val, val >> 2);
  }
```

EXECUTAREA UNEI ROTAȚII PE BIȚI

C/C++ 95

În secțiunea 94, ați învățat cum să utilizați operatorii C de deplasare la stânga și la dreapta pe biți. Când executați o operație de deplasare la stânga, C introduce zerouri în bitul cel mai puțin semnificativ. Pe de altă parte, când executați o operație de deplasare la dreapta, valoarea pe care limbajul C o pune în poziția bitului celui mai semnificativ depinde de tipul variabilei și de valoarea ei curentă. Când lucrați la nivel de bit, se poate ivi o situație în care să doriți pur și simplu să rotiți biții în loc să-i deplasați la stânga sau la dreapta. Când rotiți biții la stânga, cel mai semnificativ bit al valorii devine cel mai puțin semnificativ, în timp ce biții ceilalți se deplasează cu o poziție spre stânga. Când rotiți biții spre dreapta, bitul cel mai puțin semnificativ al valorii devine cel mai semnificativ. Ca să puteți roti biții, multe compilatoare de C pun la dispoziție funcțiile *_rotl* și *_rotr*, care rotesc biții unei valori *unsigned* la stânga sau la dreapta, în felul următor:

```
#include <stdlib.h>

unsigned _rotl(unsigned val, int contor);
unsigned _rotr(unsigned val, int contor);
```

Variabila *contor* specifică numărul de rotații pe care le efectuează valoarea. Următorul program, *rotit.c*, ilustrează utilizarea funcțiilor *_rotl* și *_rotr*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    unsigned val = 1;

    printf("%u rotit la dreapta o data este %u\n", val,
      _rotr(val, 1));
    val = 5;
    printf("%u rotit la dreapta de doua ori este %u\n", val,
      _rotr(val, 2));
    val = 65534;
    printf("%u rotit la stanga de doua ori este %u\n", val,
      _rotl(val, 2));
  }
```

Atunci când compilați și executați programul *rotit.c*, pe ecran se vor afișa următoarele:

```
1 rotit la dreapta o data este 32768
5 rotit la dreapta de doua ori este 16385
65534 rotit la stanga de doua ori este 65531
C:\>
```

***Observație:** multe compilatoare de C furnizează și funcțiile **_lrotl** și **_lrotr**, care rotesc la stânga sau la dreapta valori întregi de tipul **unsigned long**.*

96 OPERATORII CONDIȚIONALI



Toate programele prezentate până acum în această carte au început execuția cu prima instrucțiune din *main* și apoi au continuat în ordine, cu instrucțiunile următoare. Pe măsură ce programele devin mai complexe, pot apărea situații în care programul să necesite efectuarea unui anumit set de instrucțiuni dacă o condiție este adevărată, iar dacă nu este adevărată, probabil, executarea altor instrucțiuni. De exemplu, s-ar putea ca programul dumneavoastră să necesite executarea unor instrucțiuni diferite pentru fiecare din zilele săptămânii. Când un program execută (sau nu) anumite instrucțiuni în funcție de o anumită condiție, se spune că programul realizează o *procesare condițională*. Pentru a realiza o *procesare condițională*, programul evaluează o condiție returnând un rezultat adevărat sau fals. De exemplu, condiția *Astăzi este luni* este ori adevărată, ori falsă. Pentru a ajuta programele să efectueze procesarea condițională, limbajul C dispune de instrucțiunile *if*, *if-else* și *switch*. Câteva dintre capitolele care urmează vor analiza aceste instrucțiuni în detaliu.

97 PROCESAREA ITERATIVĂ



Toate programele prezentate până acum în această carte și-au executat instrucțiunile doar o singură dată. Uneori, este necesar ca programele să execute sau nu un set de instrucțiuni în funcție de rezultatul unei condiții. Pe măsură ce programele dumneavoastră devin mai complexe, pot apărea situații în care un program să repete același set de instrucțiuni de un anumit număr de ori sau până când se îndeplinește o anumită condiție. De exemplu, dacă scrieți un program care calculează calificativele studenților, programul trebuie să urmeze aceiași pași pentru fiecare student din grupă. În mod similar, dacă un program afișează conținutul unui fișier, programul va citi și va afișa fiecare linie a fișierului până când programul va întâlni simbolul de sfârșit de fișier. Atunci când programele repetă una sau mai multe instrucțiuni până când este îndeplinită o condiție dată, se spune că programul execută o *procesare iterativă*. O trecere a programului prin instrucțiunile care se repetă se numește *iterație*. Pentru a vă ajuta să realizați o procesare iterativă, limbajul C dispune de instruc- țiunile *for*, *while* și *do-while*. Unele capitole din această carte vor analiza aceste instrucțiuni în detaliu.

98 REPREZENTAREA VALORILOR ADEVĂRAT ȘI FALS



Câteva dintre secțiunile prezentate în acest capitol au analizat structurile condiționale și iterative ale limbajului C, care execută un set de instrucțiuni dacă o condiție este adevărată și, eventual, alt set de instrucțiuni dacă acea condiție este falsă. Când lucrați cu structuri *condiționale* și *iterative*, este important să înțelegeți cum reprezintă limbajul C valorile fals și adevărat. Limbajul C interpretează orice valoare diferită de 0 ca adevărat, iar valoarea 0 ca fals. De aceea, următoarea condiție va fi întotdeauna evaluată ca adevărată:

```
if (1)
```

Mulți programatori neexperimentați își scriu condițiile de test așa:

```
if (expresie != 0)   // Verifica daca expresia este adevarata
```

Atunci când doriți să testați dacă o condiție este adevărată, scrieți, pur și simplu, expresia inclusă între paranteze, ca mai jos:

```
if (expresie)
```

Când expresia este evaluată la o valoare diferită de 0 (adevărat), atunci C execută instrucțiunile care urmează imediat după condiție. Când expresia este evaluată ca 0 (fals), C nu va executa instrucțiunile care urmează imediat după condiție. Operatorii care lucrează cu valori adevărat și fals se numesc operatori *booleeni*. Rezultatul unei expresii booleene este întotdeauna adevărat sau fals.

TESTAREA UNEI CONDIȚII CU IF

C/C++ 99

Pe măsură ce programele dumneavoastră vor deveni mai complexe, ele vor efectua adesea un anumit set de instrucțiuni atunci când o condiție este adevărată și un alt set de instrucțiuni atunci când acea condiție este falsă. Când programul dumneavoastră trebuie să execute o astfel de procesare condițională, veți utiliza instrucțiunea *if*. Formatul acestei instrucțiuni este:

```
if (conditie)
   instructiune;
```

Condiția pe care o evaluează instrucțiunea *if* trebuie să apară între paranteze și trebuie să fie sau adevărată, sau falsă. Atunci când condiția este adevărată, se va executa instrucțiunea care urmează imediat după condiție. De exemplu, următoarea instrucțiune *if* testează dacă variabila *varsta* este mai mare sau egală cu 21. Dacă condiția este adevărată, programul va executa instrucțiunea *printf*. Dacă condiția este falsă, programul nu va executa instrucțiunea *printf* și va continua execuția începând cu prima instrucțiune care urmează după *printf* (instrucțiunea care atribuie valoarea 185 variabilei *inaltime*):

```
if (varsta >= 21)
   printf("Variabila varsta este 21 sau mai mult\n");
inaltime = 185;
```

INSTRUCȚIUNILE SIMPLE ȘI CELE COMPUSE

C/C++ 100

Când programul dumneavoastră efectuează o procesare condițională, uneori se întâmplă ca programul să execute una sau mai multe instrucțiuni dacă o condiție este adevărată și, eventual, alte instrucțiuni dacă este falsă. De asemenea, atunci când programul dumneavoastră efectuează o procesare iterativă, uneori el va repeta o singură instrucțiune, alteori el va repeta câteva instrucțiuni. În cadrul procesărilor *iterative* sau *condiționale*, limbajul C consideră instrucțiunile ca fiind simple sau compuse. O *instrucțiune simplă* este o singură instrucțiune, cum ar fi aceea prin care se atribuie o valoare unei variabile sau se apelează

funcția *printf*. Următoarea instrucțiune *if* apelează o instrucțiune simplă (*printf*) atunci când condiția este adevărată:

```
if (conditie)
   printf("Conditia este adevarata\n");
```

Pe de altă parte, o *instrucțiune compusă* este alcătuită din două sau mai multe instrucțiuni incluse între acolade. Următorul exemplu de utilizare a instrucțiunii *if* conține o instrucțiune compusă:

```
if (conditie)
   {
    varsta = 21;
    inaltimea = 185;
    greutatea = 75;
   }
```

Când programul dumneavoastră trebuie să efectueze mai multe instrucțiuni pornind de la o anumită condiție sau atunci când trebuie să repete câteva instrucțiuni, veți utiliza o instrucțiune compusă, plasând aceste instrucțiuni între acolade.

101 TESTAREA UNEI EGALITĂȚI

C/C++

Pe măsură ce programele dumneavoastră vor deveni mai complexe, va apărea necesitatea de a compara valoarea unei variabile cu o anumită condiție pentru a stabili ce instrucțiuni vor fi executate în continuare. Aceste decizii pot fi implementate cu instrucțiunile *if* sau *switch*. Așa cum ați învățat în secțiunea 99, formatul unei instrucțiuni *if* este :

```
if (conditie)
   instructiune;
```

Cele mai multe instrucțiuni *if* vor testa dacă valoarea unei variabile este egală cu o anumită valoare specificată. De exemplu, următoarea instrucțiune *if* testează dacă variabila *varsta* are valoarea 21:

```
if (varsta == 21)
   instructiune;
```

Pentru a testa o egalitate, limbajul C utilizează semnul egal dublu (==). Atunci când introduceți în program o astfel de testare, utilizați semnul egal dublu (==) în loc de semnul egal simplu (=), pe care limbajul C îl utilizează pentru operația de atribuire. Așa cum veți învăța în secțiunea 112, dacă utilizați *operatorul de atribuire* (=) în locul semnului egal dublu, compilatorul va considera condiția ca fiind corectă din punct de vedere sintactic, dar, din păcate, nu va testa dacă variabila are acea valoare. În schimb, va atribui acea valoare variabilei.

Observație: *În funcție de nivelul de avertizare stabilit pentru compilatorul dumneavoastră, poate fi afișat un mesaj care să semnaleze realizarea unei operații de atribuire în locul testării condiției.*

Așa cum programele dumneavoastră trebuie să testeze uneori o condiție de egalitate, tot așa ele vor trebui câteodată să testeze o anumită inegalitate. Limbajul C utilizează simbolul !=

pentru a testa o inegalitate. Următoarea instrucțiune testează dacă variabila *varsta* este diferită de 21:

```
if (varsta != 21)
   instructiune;
```

Următorul program *eg_ineg.c* utilizează testările de egalitate (==) și cele de inegalitate (!=):

```
#include <stdio.h>

void main(void)
  {
    int varsta = 21;
    int inaltime = 75;

    if (varsta == 21)
      printf("Varsta utilizatorului este 21\n");
    if (varsta != 21)
      printf("Varsta utilizatorului nu este 21\n");
    if (inaltime == 75)
      printf("Inaltimea utilizatorului este 75\n");
    if (inaltime != 75)
      printf("Inaltimea utilizatorului nu este 75\n");
  }
```

Atunci când compilați și executați acest program, pe monitor va apărea următorul rezultat:

```
Varsta utilizatorului este 21
Inaltimea utilizatorului este 75
C:\>
```

Pentru a înțelege modul de utilizare a operatorilor de egalitate și de inegalitate, testați programul *eg_ineg.c* modificând valorile variabilelor *inaltime* și *varsta*.

EXECUTAREA TESTELOR RELAȚIONALE

C/C++ 102

Pe măsură ce programele dumneavoastră vor deveni mai complexe, veți avea nevoie să testați dacă o valoare este mai mare, mai mică, mai mare sau egală sau dacă este mai mică sau egală cu o altă valoare. Pentru a vă ajuta să realizați această testare, limbajul C conține un set de *operatori relaționali*. Tabelul 102 prezintă lista operatorilor relaționali ai limbajului C

Operator	Funcție
>	Operatorul *mai mare decât*
<	Operatorul *mai mic decât*
>=	Operatorul *mai mare sau egal cu*
<=	Operatorul *mai mic sau egal cu*

Tabelul 102 *Operatorii relaționali ai limbajului C.*

Următoarea instrucțiune *if* utilizează operatorul C *mai mare sau egal cu* (>=) pentru a testa dacă variabila *varsta* este mai mare decât 20:

```
if (varsta >= 21)
   printf("Varsta utilizatorului este mai mare de 20\n");
```

103 UTILIZAREA OPERATORULUI ȘI LOGIC PENTRU TESTAREA A DOUĂ CONDIȚII

C/C++

În secțiunea 99, ați învățat cum se utilizează instrucțiunea *if* pentru a testa condițiile într-un program. Pe măsură ce programele dumneavoastră vor deveni mai complexe, veți testa mai multe condiții în cadrul lor. De exemplu, poate că veți dori ca instrucțiunea *if* să testeze dacă utilizatorul are un câine și, în caz că are, dacă acest câine este dalmațian. În cazurile în care vreți să testați dacă două condiții sunt adevărate, folosiți operatorul *ŞI logic*. Limbajul C reprezintă operatorul *ŞI logic* prin doua caractere &&, așa cum se vede mai jos:

```
if ((utilizatorul_are_caine) && (caine == dalmatian))
   {
     // Instructiuni
   }
```

Când compilatorul de C întâlnește o instrucțiune *if* care utilizează operatorul *ŞI logic* (&&), evaluează cele două condiții de la stânga spre dreapta. Dacă examinați parantezele, veți observa că instrucțiunea *if* de mai sus are forma următoare:

```
if (conditie)
```

În următorul exemplu, condiția este compusă de fapt din două condiții conectate cu operatorul *ŞI logic*:

```
(utilizatorul_are_caine) && (caine == dalmatian)
```

Pentru a fi adevărată condiția realizată cu operatorul *ŞI logic*, trebuie ca ambele condiții să fie evaluate ca adevărate. Dacă vreuna dintre ele este falsă, condiția rezultată este evaluată ca falsă.

Multe secțiuni prezentate în această carte vor utiliza operatorul *ŞI logic*. De fiecare dată, condițiile vor fi plasate între paranteze pentru a se asigura evaluarea expresiilor cu o precedență corectă a operatorilor.

Observație: *Să nu confundați operatorul* ***ŞI logic (&&)*** *cu operatorul* ***ŞI pe biți (&)***. *Operatorul* ***ŞI logic*** *evaluează două expresii* ***booleene*** *(care pot avea valoarea adevărat sau fals) pentru a produce un rezultat adevărat sau fals. În schimb, operatorul* ***ŞI pe biți***, *manipulează biții (care au valoarea 1 sau 0).*

104 UTILIZAREA OPERATORULUI SAU LOGIC PENTRU TESTAREA A DOUĂ CONDIȚII

C/C++

În secțiunea 99, ați învățat cum se utilizează instrucțiunea *if* pentru a testa condițiile într-un program. Pe măsură ce programele dumneavoastră vor deveni mai complexe, poate că veți testa multe condiții. De exemplu, poate veți dori ca instrucțiunea *if* să testeze dacă

utilizatorul are un câine sau dacă utilizatorul are un calculator. În cazurile când vreți să testați dacă una dintre cele două condiții este adevărată (ori dacă sunt adevărate amândouă), puteți să folosiți operatorul *SAU logic*. Limbajul C reprezintă operatorul *SAU logic* prin doua bare verticale (| |), așa cum se vede mai jos:

```
if ((utilizatorul_are_caine) || (utilizatorul_are_calculator))
  {
    // Instructiuni
  }
```

Când compilatorul de C întâlnește o instrucțiune *if* care utilizează operatorul *SAU logic* (| |), evaluează cele două condiții de la stânga spre dreapta. Dacă examinați parantezele, veți observa că instrucțiunea *if* de mai sus are forma următoare:

```
if (conditie)
```

În acest exemplu particular, este compusă de fapt din două condiții conectate cu operatorul *SAU logic*:

```
(utilizatorul_are_caine) || (utilizatorul_are_calculator)
```

Pentru a fi adevărată condiția realizată cu operatorul *SAU logic*, trebuie ca numai una dintre condiții să fie evaluată ca adevărată. Dacă vreuna dintre ele este adevărată (sau amândouă sunt adevărate), condiția rezultată este evaluată ca adevărată. Dacă ambele condiții sunt evaluate ca false, atunci condiția rezultată este falsă.

Multe secțiuni prezentate în această carte utilizează operatorul *SAU logic* (| |). De fiecare dată, condițiile vor fi plasate între paranteze pentru a se asigura evaluarea expresiilor cu o precedență corectă a operatorilor.

Observație: *Să nu confundați operatorul* ***SAU logic*** *(| |) cu operatorul* ***SAU pe biți*** *(|). Operatorul* ***SAU logic*** *evaluează două expresii* ***booleene*** *(care pot avea valoarea adevărat sau fals) pentru a produce un rezultat adevărat sau fals. În schimb, operatorul* ***SAU pe biți****, manipulează biții (care au valoarea 1 sau 0).*

EXECUTAREA OPERAȚIEI NU LOGIC

C/C++ 105

Când programul dumneavoastră utilizează instrucțiunea *if* pentru a executa o procesare condițională, instrucțiunea *if* evaluează o expresie care va produce un rezultat adevărat sau fals. În funcție de necesitățile programului dumneavoastră, pot apărea situații în care să doriți execuția unui set de instrucțiuni când condiția este evaluată ca falsă. De exemplu, să presupunem că doriți ca programul să testeze dacă utilizatorul are un câine. Dacă utilizatorul nu are un câine, programul trebuie să afișeze un mesaj care îi spune utilizatorului să-și cumpere un dalmațian. Dacă utilizatorul are un câine, programul nu trebuie să facă nimic. Atunci când doriți ca programul să execute una sau mai multe instrucțiuni în cazul unei condiții evaluate ca falsă, veți utiliza operatorul *NU logic*, pe care C îl reprezintă folosind semnul exclamării (!). Să analizăm următoarea instrucțiune *if*:

```
if (! utilizatorul_are_caine)
  printf("Trebuie sa cumparati un dalmatian\n");
```

Condițiile care conțin operatorul *NU logic* spun în esență că atunci când o condiție nu este adevărată (cu alte cuvinte, condiția este evaluată ca falsă), trebuie să se execute instrucțiunile (sau instrucțiunile compuse). În multe dintre secțiunile prezentate în această carte apar condiții care conțin operatorul *NU logic*.

106 ATRIBUIREA REZULTATULUI EVALUĂRII UNEI CONDIȚII



Câteva secțiuni din acest capitol v-au prezentat diferite condiții care sunt evaluate ca adevărate sau false în cadrul unei instrucțiuni *if, while, for* sau de alt tip. Limbajul C nu numai că vă permite să utilizați condiții în cadrul structurilor de control condiționale sau iterative, dar vă permite, de asemenea, să atribuiți unei variabile rezultatul evaluării unei condiții. Să presupunem, de exemplu, că programul dumneavoastră utilizează rezultatul evaluării aceleiași condiții de mai multe ori, ca mai jos:

```
if ((strlen(nume) < 100) && (azi == LUNI))
  {
    // Instructiuni
  }
else if ((strlen(nume) < 100) && (azi == MARTI))
  {
    // Instructiuni
  }
else if (strlen(nume) >= 100)
  {
    // Instructiuni
  }
```

După cum vedeți, programul utilizează condiția *(strlen(nume) < 100)* de trei ori. De fiecare dată când apare această condiție, programul apelează funcția *strlen*. În instrucțiunile precedente, programul poate apela de trei ori funcția *strlen* (în funcție de valoarea variabilei *azi*). Următoarele instrucțiuni vor atribui variabilei *nume_ok* rezultatul evaluării condiției (adevărat sau fals) și apoi va utiliza în mod repetat această variabilă în locul condiției. Folosind variabila în locul condiției, așa cum se vede mai jos, performanțele programului vor crește.

```
nume_ok = (strlen(nume) < 100);
if (nume_ok && (azi == LUNI))
  {
    // Instructiuni
  }
else if (nume_ok && (azi == MARTI))
  {
    // Instructiuni
  }
else if (! nume_ok)
  {
    // Instructiuni
  }
```

DECLARAREA VARIABILELOR ÎN CADRUL INSTRUCȚIUNILOR COMPUSE

C/C++ 107

În secțiunea 100, ați văzut care este diferența dintre instrucțiunile simple și cele compuse. Așa cum ați învățat, o instrucțiune compusă este alcătuită din una sau mai multe instrucțiuni încadrate de acolade ({}). Următorul exemplu de buclă *while* (care va citi liniile dintr-un fișier și le va afișa cu majuscule) conține o instrucțiune compusă:

```
while (fgets(linie, sizeof(linie), pointer_fisier))
  {
    strupr(linie);
    fputs(linie, stdout);
  }
```

Pe măsură ce programele dumneavoastră vor deveni mai complexe, uneori procesarea din cadrul unei instrucțiuni compuse va necesita utilizarea uneia sau mai multor variabile ale căror valori le veți folosi numai în cadrul acelei bucle (cum ar fi cazul variabilelor contor). De exemplu, atunci când utilizați variabile contor, în general veți declara aceste variabile la începutul programului, imediat după instrucțiunea *main*. Totuși, dacă utilizați o variabilă numai în interiorul unei instrucțiuni compuse, puteți declara variabila la începutul instrucțiunii, așa cum se vede mai jos:

```
if (conditie)
  {
    int contor;
    float total;
    // Alte instructiuni
  }
```

În acest caz, programul declară două variabile la începutul instrucțiunii compuse. Puteți utiliza aceste două variabile în cadrul instrucțiunii compuse, ca și cum le-ați fi definit la începutul programului. Însă nu veți putea referi aceste valori în afara acoladelor care includ instrucțiunea compusă. Avantajul declarării variabilelor în cadrul unei instrucțiuni compuse este că un alt programator, când vă citește programul, va înțelege mai bine cum și unde utilizați o variabilă. În unele dintre capitolele următoare, vom aborda noțiunea de domeniu de vizibilitate a unei variabile sau, altfel spus, zona din program în care acesta recunoaște o variabilă. Ca regulă, ar trebui să faceți în așa fel încât programul să recunoască o variabilă numai în locațiile în care este efectiv folosită. Cu alte cuvinte trebuie limitată vizibilitatea variabilei. Declararea variabilelor la începutul unei instrucțiuni compuse, așa cum s-a arătat în această secțiune, limitează domeniul de vizibilitate al variabilei la spațiul inclus între acoladele de la începutul și sfârșitul instrucțiunii compuse.

***Observație**: Dacă declarați în cadrul unei instrucțiuni compuse variabile care au același nume ca și variabilele definite în afara instrucțiunii compuse, compilatorul de C va utiliza variabilele nou declarate în interiorul instrucțiunii compuse și variabilele inițiale în afara ei.*

108 FOLOSIREA INDENTĂRII PENTRU ÎMBUNĂTĂȚIREA LIZIBILITĂȚII

Una dintre cele mai bune metode de mărire a lizibilității programului este utilizarea indentării. De fiecare dată când programul dumneavoastră folosește o acoladă (cum ar fi începutul unei instrucțiuni compuse), ar trebui să indentați liniile care urmează cu unul sau două spații. De exemplu, să considerăm următorul program, *cu_ind.c*:

```
#include <stdio.h>
void main(void)
  {
    int varsta = 10;
    int utilizatorul_are_caine = 0; // 0 este fals
    if (varsta == 10)
      {
        printf("Cainele este prietenul omului\n");
        if (!utilizatorul_are_caine)
          printf("Cumpara un dalmatian\n");
      }
    printf("Grivei este dalmatian\n");
  }
```

Nu este nevoie decât să priviți indentarea pentru a vă da imediat seama care este structura generală a programului, de exemplu care sunt instrucțiunile compuse. Indentarea este lipsită de importanță pentru compilator. Acesta nu face deosebire între programul precedent și următorul program, *nu_ind.c*:

```
#include <stdio.h>
void main(void)
{
int varsta = 10;
int utilizatorul_are_caine = 0; // 0 este fals
if (varsta == 10)
{
printf("Cainele este prietenul omului\n");
if (!utilizatorul_are_caine)
printf("Cumpara un dalmatian\n");
}
printf("Grivei este dalmatian\n");
}
```

După cum puteți vedea, indentarea face ca primul program să fie mult mai ușor de înțeles pentru dumneavoastră sau alt cititor.

UTILIZAREA TESTĂRII EXTINSE A COMBINAȚIEI CTRL+BREAK

C/C++ 109

Atunci când creați programe care utilizează bucle de tipul *for*, *while* sau *do* pentru iterații în mediul DOS, pot apărea situații în care va trebui să folosiți combinația de taste *CTRL+BREAK* pentru a termina un program care a intrat într-o buclă infinită. În mod prestabilit, mediul DOS verifică dacă a fost tastată combinația CTRL+BREAK după fiecare scriere pe ecran, pe disc sau la imprimantă sau atunci când citește un caracter de la tastatură. Dacă programul dumneavoastră nu execută nici o operație de acest gen în cadrul buclei pe care doriți să o întrerupeți, nu veți putea utiliza comanda CTRL+BREAK pentru a termina execuția programului. Totuși, utilizând comanda BREAK a mediului DOS, puteți mări numărul operațiilor la sfârșitul cărora DOS verifică dacă a apărut o intrare CTRL+BREAK. Această verificare suplimentară este denumită de programatori *testare extinsă a combinației CTRL+BREAK*. Următoarea comandă BREAK activează testarea extinsă a combinației CTRL+BREAK:

```
C:\> BREAK ON <ENTER>
```

Dacă doriți ca mediul DOS să activeze automat testarea extinsă a combinației CTRL+BREAK imediat după pornirea sistemului, plasați o intrare BREAK=ON în fișierul dumneavoastră *config.sys*. Deoarece mediul DOS efectuează mai multe testări ale combinației CTRL+BREAK, performanțele generale ale sistemului dumneavoastră vor descrește puțin. Totuși, când veți realiza primele procesări iterative, s-ar putea să considerați că posibilitatea de a încheia un program cu ajutorul combinației CTRL+BREAK este mai importantă decât ușoara scădere a performanțelor.

TESTAREA VALORILOR ÎN VIRGULĂ MOBILĂ

C/C++ 110

Câteva capitole prezentate în această secțiune au folosit instrucțiunile *if* și *while* pentru testa valoarea variabilelor. De exemplu, următoarele instrucțiuni testează câteva variabile întregi:

```
if (varsta == 21)
   // Instructiuni
if (inaltime > 173)
   // Instructiuni
```

Când lucrați cu valori în virgulă mobilă, trebuie să fiți precaut la testarea valorii variabilei. De exemplu, următoarea instrucțiune testează valoarea unei variabile numită *impozit_vanzari*.

```
if (impozit_vanzari == 0.065)
   // Instructiuni
```

În secțiunea 51, ați învățat despre precizia valorilor în virgulă mobilă și despre faptul că un calculator trebuie să reprezinte valorile în virgulă mobilă utilizând un număr precizat de biți. Este imposibil pentru calculatorul dumneavoastră să reprezinte exact orice valoare. În cazul instrucțiunii *if* precedente, de exemplu, calculatorul poate să reprezinte valoarea 0.065 ca 0.0649999. Prin urmare, instrucțiunea *if* nu va fi niciodată evaluată ca adevărată. Pentru a preveni o asemenea eroare în programele dumneavoastră, nu trebuie să testați valori în virgulă mobilă exacte, ci intervale acceptabile de valori, ca mai jos:

```
if (impozit_vanzari - 0.065)<= 0.0001
  // Instructiuni
```

În exemplul precedent, când diferența între valoarea variabilei *impozit_vanzari* și 0.065 este mai mică sau egală cu 0.0001, programul va considera că valorile sunt egale.

111 BUCLAREA INFINITĂ



Așa cum ați învățat, instrucțiunile *for, while* și *do while* vă permit să repetați una sau mai multe instrucțiuni până când este întâlnită condiția dată. În funcție de programul dumneavoastră, este posibil ca programul să facă o buclare infinită. De exemplu, un program care detectează scurgerile de radiații într-un reactor nuclear ar trebui să ruleze în continuu. Pentru a face ca programul dumneavoastră să repete continuu o buclă, pur și simplu plasați o constantă diferită de 0 în interiorul buclei, ca mai jos:

```
while (1)
```

Atunci când utilizați o valoare diferită de 0 pentru a obliga programul dumneavoastră să execute continuu o buclă, puteți să definiți constanta pentru a îmbunătăți lizibilitatea programului. De exemplu, puteți utiliza constanta *FOREVER*, așa cum se vede mai jos:

```
#define FOREVER 1
while (FOREVER)
```

Pentru a crea o buclă pentru reactorul nuclear din exemplul precedent, puteți proceda în felul următor:

```
#define TOPIRE 0
while (!TOPIRE)
```

112 TESTAREA UNEI ATRIBUIRI



După cum ați învățat, limbajul C utilizează semnul egal ca operator de *atribuire* și semnul egal dublu (==) ca test al egalității, așa cum se vede în exemplul de mai jos:

```
punctaj = 100;
if (punctaj == MAX)
   {
     // Instructiuni
   }
```

În fragmentul de cod de mai sus, prima instrucțiune atribuie valoarea 100 variabilei *punctaj*, apoi instrucțiunea *if* testează valoarea variabilei. Pentru a vă ajuta să reduceți numărul de instrucțiuni din program, limbajul C vă permite să testați rezultatul unei atribuiri. De exemplu, următoarea instrucțiune *if* combină operația de atribuire cu testarea condiției:

```
if ((punctaj = 100) == MAX)
   {
```

```
    // Instructiuni
  }
```

Compilatorul de C va efectua mai întâi expresia conținută între paranteze, atribuind valoarea 100 variabilei *punctaj*. Apoi el va compara valoarea pe care ați atribuit-o variabilei *punctaj* cu constanta *MAX*. Dacă scoateți parantezele, ca mai jos, se va atribui o altă valoare și testul va fi diferit:

```
if (punctaj = 100 == MAX)
```

Fără paranteze, compilatorul de C testează dacă valoarea 100 este egală cu constanta *MAX* și, dacă este așa, atribuie valoarea 1 (adevărat) variabilei *punctaj*. Dacă valoarea *MAX* nu este egală cu 100, instrucțiunea atribuie valoarea 0 (fals) variabilei *punctaj*.

De cele mai multe ori, veți utiliza testarea rezultatului unei atribuiri atunci când veți dori să testați valoarea pe care o returnează o funcție (cum ar fi *fopen* sau *getchar*), ca în exemplul de mai jos:

```
if ((pointer_fisier = fopen("CONFIG.SYS", "r")) == NULL)
  {
    // Instructiuni
  }
if ((litera = getchar()) == 'A')
  {
    // Instructiuni
  }
```

UTILIZAREA INSTRUCȚIUNILOR IF-IF-ELSE

C/C++ 113

Atunci când utilizați instrucțiuni *if-else*, o eroare logică perfidă vă poate face probleme dacă nu țineți seama cărei instrucțiuni *if* îi corespunde fiecare *else*. De exemplu, să considerăm următorul fragment de cod:

```
punctaj_test = 100;
calificativ_curent = 'B';

if (punctaj_test >= 90)
   if (calificativ_curent == 'A')
     printf("Un alt A pentru un student cu calificativ A\n");
else
   printf("Trebuia sa fi muncit mai mult\n");
```

Prima instrucțiune *if* testează dacă punctajul la test al unui student a fost mai mare sau egal cu 90. Dacă e așa, o a doua instrucțiune *if* testează dacă studentul are deja calificativul A și, în acest caz, afișează un mesaj. Bazându-vă pe indentare, v-ați aștepta ca instrucțiunea *else* să-și afișeze mesajul dacă punctajul la test a fost mai mic decât 90. Din păcate, nu acesta este modul în care fragmentul de cod prelucrează condițiile. Atunci când plasați o instrucțiune *else* în cadrul programului, acest *else* va fi asociat cu cea mai apropiată instrucțiune *if* care nu are asociat un *else*. Cu toate că punctajul la test al studentului a fost 100, fragmentul de cod de mai sus ar putea afișa mesajul care spune studentului că trebuia să fi muncit mai mult. Cu alte cuvinte fragmentul execută instrucțiunile de mai jos:

```
if (punctaj_test >= 90)
   if (calificativ_curent == 'A')
     printf("Un alt A pentru un student cu calificativ A\n");
   else
     printf("Trebuia sa fi muncit mai mult\n");
```

Pentru a evita asocierea instrucțiunii *else* cu o instrucțiune *if* nepotrivită, plasați a doua instrucțiune *if* între acolade, formând o *instrucțiune compusă*, așa cum se vede mai jos:

```
if (punctaj_test >= 90)
   {
     if (calificativ_curent == 'A')
     printf("Un alt A pentru un student cu calificativ A\n");
   }
else
   printf("Trebuia sa fi muncit mai mult\n");
```

114 *Efectuarea unor instrucțiuni de un anumit număr de ori*



Una dintre operațiile pe care programele dumneavoastră le vor efectua frecvent este repetarea unui set de instrucțiuni de un anumit număr de ori. De exemplu, poate că veți dori să calculați punctajul la test pentru 30 de studenți, să determinați care sunt cele mai mari și cele mai mici valori într-un set de 100 de cotații la bursă sau chiar să puneți calculatorul să emită de trei ori un sunet. Pentru a ajuta programele dumneavoastră să repete una sau mai multe instrucțiuni de un anumit număr de ori, limbajul C prevede oferă instrucțiunea *for*. Sintaxa instrucțiunii *for* este următoarea:

```
for (valoare_start; conditie_sfarsit; valoare_increment)
   instructiune;
```

Atunci când programul repetă (ciclează) instrucțiuni de un anumit număr de ori, veți folosi de obicei o variabilă denumită *variabilă de control*, care indică de câte ori ați efectuat aceste instrucțiuni. Instrucțiunea *for* conține patru secțiuni. Secțiunea *valoare_start* atribuie variabilei de control o valoare inițială, care, de cele mai multe ori, este 0 sau 1. Secțiunea *condiție_sfarsit* testează, de obicei, valoarea variabilei de control pentru a stabili dacă programul a executat instrucțiunile de atâtea ori cât s-a dorit. Secțiunea *valoare_increment* adaugă, de obicei, valoarea 1 la variabila de control de fiecare dată când se execută instrucțiunile. În sfârșit, a patra secțiune a instrucțiunii *for* este reprezentată de instrucțiunea sau instrucțiunile pe care doriți să le repetați. Deoarece programul dumneavoastră efectuează în mod repetat instrucțiunea sau instrucțiunile specificate (se întoarce la începutul instrucțiunii), instrucțiunea *for* este de multe ori denumită *buclă for*. Să luăm de exemplu următoarea instrucțiune *for*, care va afișa pe ecran numerele de la 1 la 10:

```
for (contor = 1; contor <= 10; contor++)
   printf("%d\n", contor);
```

În exemplul precedent, *contor* este variabila de control a buclei. Mai întâi, instrucțiunea *for* atribuie valoarea 1 acestei variabile, după care testează imediat dacă valoarea variabilei *contor* este mai mică sau egală cu 10 (condiția de sfârșit a buclei). Dacă variabila *contor* are o

valoare mai mică sau egală cu 10, bucla *for* va efectua imediat următoarea instrucțiune, care, în exemplul dat, este *printf*. După ce programul a efectuat instrucțiunea *printf*, bucla *for* execută expresia specificată în secțiunea *valoare_increment*. În cazul de față, bucla *for* incrementează valoarea variabilei *contor* cu 1, apoi bucla *for* testează imediat condiția de sfârșit. Dacă valoarea variabilei *contor* este mai mică sau egală cu 10, bucla se repetă. De aceea, instrucțiunea *printf* va afișa 1 la prima executare a buclei. La a doua execuție a buclei, valoarea variabilei *contor* este 2, apoi 3 și așa mai departe. După ce *printf* afișează valoarea 10, secțiunea *valoare_increment* crește valoarea *contor*, făcând-o 11. Când bucla *for* efectuează testarea condiției de sfârșit, ea va observa că valoarea variabilei contor nu mai este mai mică sau egală cu 10 și astfel bucla se va sfârși, iar programul dumneavoastră își va relua prelucrarea începând cu prima instrucțiune care urmează după bucla *for*.

Pentru a înțelege mai bine structura buclei *for*, să considerăm următorul program, *test_for.c*:

```
#include <stdio.h>

void main(void)
  {
    int contor;

    for (contor = 1; contor <= 5; contor++)
      printf("%d ", contor);
    printf("\nIncepe a doua bucla\n");
    for (contor = 1; contor <= 10; contor++)
      printf("%d ", contor);
    printf("\nIncepe a treia bucla\n");
    for (contor = 100; contor <= 5; contor++)
      printf("%d ", contor);
  }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră se vor afișa următoarele:

```
1 2 3 4 5
Incepe a doua bucla
1 2 3 4 5 6 7 8 9 10
Incepe a treia bucla
C:\>
```

După cum puteți vedea, prima buclă *for* afișează numerele de la 1 la 5. A doua buclă *for* afișează numerele de la 1 la 10. A treia buclă *for* nu afișează nici o valoare. Dacă urmăriți cu atenție, veți vedea că programul atribuie la început valoarea 100 variabilei de control. Atunci când instrucțiunea *for* testează valoarea, bucla va întâlni imediat condiția de sfârșit, deci bucla nu se va executa.

Toate exemplele prezentate în această secțiune au utilizat o singură instrucțiune în bucla *for*. Dacă trebuie să repetați mai multe instrucțiuni, plasați instrucțiunile între acolade, formând o *instrucțiune compusă*, ca mai jos:

```
for (i = 1; i <= 10; i++)
  {
    // Instructiuni
  }
```

115 PĂRȚILE OPȚIONALE ALE INSTRUCȚIUNII FOR



În secțiunea 114, ați învățat că instrucțiunea *for* permite programului dumneavoastră să repete una sau mai multe instrucțiuni de un anumit număr de ori. Așa cum ați învățat, instrucțiunea *for* are trei secțiuni: o inițializare, un test și o incrementare (a patra secțiune conține instrucțiunile pe care le repetă bucla *for*):

```
for (initializare; test; incrementare)
```

În funcție de programele dumneavoastră, probabil că nu veți avea nevoie întotdeauna de toate cele trei secțiuni ale instrucțiunii *for*. De exemplu, dacă ați atribuit mai înainte variabilei *contor* valoarea inițială 0, puteți sări peste secțiunea de inițializare a buclei. Într-o astfel de situație, pentru a afișa numerele de la 0 la 999, bucla dumneavoastră va conține următoarele:

```
for (; contor < 1000; contor++)
   printf(" %d", contor);
```

Dacă omiteți una dintre secțiunile buclei *for*, trebuie totuși să includeți punctul și virgula aferente. De exemplu, următoarea buclă *for* sare peste secțiunile de inițializare și de incrementare:

```
for (; contor < 1000; )
   printf(" %d", contor++);
```

Asemănător, următoarea instrucțiune *for* va cicla la infinit:

```
for (;;)
  // Instructiune
```

Cu toate că aceste secțiuni ale instrucțiunii *for* au un caracter opțional, programul dumneavoastră va deveni foarte dificil de citit dacă le omiteți. Ca regulă, dacă nu aveți nevoie să utilizați toate cele trei părți ale instrucțiunii *for*, ar trebui să utilizați o altă structură ciclică, cum ar fi instrucțiunea *while*.

116 DECREMENTAREA VALORILOR ÎN CADRUL UNEI INSTRUCȚIUNI FOR



Așa cum ați învățat, instrucțiunea *for* vă permite să repetați una sau mai multe instrucțiuni de un anumit număr de ori. Secțiunile 114 și 115 au prezentat mai multe instrucțiuni *for*. În fiecare caz, bucla *for* număra crescător de la 1 la 5, de la 1 la 10 și așa mai departe. De asemenea, instrucțiunea *for* vă permite și decrementarea variabilei de control. De exemplu, următoarea buclă *for* afișează în ordine descrescătoare numerele de la 10 la 1:

```
for (contor = 10; contor >= 1; contor--)
   printf(" %d", contor);
```

Așa cum puteți observa, instrucțiunea *for* de mai sus este practic contrariul instrucțiunilor *for* întâlnite în capitolele precedente. Bucla inițializează variabila de control *contor* cu o valoare mai mare și apoi decrementează variabila cu 1 de fiecare dată când se repetă bucla.

Următorul program, *for_desc.c*, utilizează instrucțiunea *for* pentru a număra descrescător de la 5 la 1, apoi de la 10 la 1:

```
#include <stdio.h>

void main(void)
  {
    int contor;

    for (contor = 5; contor >= 1; contor--)
      printf("%d ", contor);
    printf("\nIncepe a doua bucla\n");
    for (contor = 10; contor >= 1; contor--)
      printf("%d ", contor);
    printf("\nIncepe a treia bucla\n");
    for (contor = 0; contor >= 1; contor--)
      printf("%d ", contor);
  }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră se vor afișa următoarele:

```
5 4 3 2 1
Incepe a doua bucla
10 9 8 7 6 5 4 3 2 1
Incepe a treia bucla
C:\>
```

Așa cum puteți vedea, a treia buclă *for* nu afișează nici o valoare. În acest exemplu, instrucțiunea *for* inițializează variabila *contor* cu o valoare care este mai mică decât valoarea finală 1, deci bucla se sfârșește imediat.

CONTROLUL INCREMENTĂRII ÎNTR-O BUCLĂ FOR

C/C++ 117

Așa cum ați învățat, bucla *for* permite programelor dumneavoastră să repete una sau mai multe instrucțiuni de un anumit număr de ori. În secțiunile anterioare, la fiecare executare a buclei *for*, valoarea variabilei de control era ori incrementată, ori decrementată cu 1. Dar limbajul C vă permite să incrementați valoarea variabilei cu orice mărime. De exemplu, următoarea buclă *for* incrementează cu 10 variabila de control *contor* la fiecare iterație:

```
for (contor = 0; contor <= 100; contor+= 10)
    printf("%d ", contor);
```

De asemenea, buclele *for* de mai sus au inițializat variabila de control cu 1 sau 0. Tot așa cum puteți incrementa sau decrementa variabila cu orice mărime doriți, limbajul C vă permite să inițializați variabila cu orice valoare. Următorul program, *for_dif.c*, utilizează valori diferite de inițializare și de decrementare:

```
#include <stdio.h>

void main(void)
  {
    int contor;

    for (contor = -100; contor <= 100; contor += 5)
```

```
    printf("%d ", contor);
   printf("\nIncepe a doua bucla\n");
   for (contor = 100; contor >= -100; contor -= 25)
     printf("%d ", contor);
 }
```

118 UTILIZAREA BUCLELOR FOR CU VALORI DE TIP CHAR ȘI FLOAT



Așa cum ați învățat, bucla *for* permite programelor dumneavoastră să repete una sau mai multe instrucțiuni de un anumit număr de ori. În secțiunile anterioare, instrucțiunile *for* au utilizat numai valori de tipul *int*, dar puteți utiliza, de asemenea, valori de tip caracter sau număr în virgulă mobilă. De exemplu, următoarea buclă *for* afișează literele alfabetului:

```
for (litera = 'A'; litera <= 'Z'; litera++)
  printf("%c", litera);
```

În mod asemănător, următoarea buclă incrementează o valoare în virgulă mobilă cu 0,5:

```
for (procent=0.0; procent <=100.0; procent +=0.5)
  printf ('' %f\n'', procent);
```

Următorul program, *for_ext.c*, ilustrează modul de utilizare a literelor sau a valorilor reale în virgulă mobilă într-o buclă *for*:

```
#include <stdio.h>

void main(void)
 {
   char litera;
   float procent;

   for (litera = 'A'; litera <= 'Z'; litera++)
     putchar(litera);
   for (litera = 'z'; litera >= 'a'; litera--)
     putchar(litera);
   putchar('\n');
   for (procent = 0.0; procent < 1.0; procent += 0.1)
     printf("%3.1f\n", procent);
 }
```

119 BUCLA VIDĂ



Așa cum ați învățat, bucla *for* vă permite să repetați una sau mai multe instrucțiuni de un anumit număr de ori, până când variabila de control a buclei îndeplinește o condiție dată. În trecut, atunci când programatorii doreau ca programele lor să facă o scurtă pauză, de exemplu pentru a afișa un mesaj, ei plasau o *buclă vidă* (o buclă care „nu face nimic") în programele lor. De exemplu, următoarea buclă nu face nimic timp de 100 de ori:

```
for (contor = 1; contor <= 100; contor ++)
    ; // Nu face nimic
```

Atunci când plasați o buclă vidă într-un program, se va efectua mai întâi inițializarea buclei și apoi va repeta testul și se va incrementa variabila de control până la îndeplinirea condiției de sfârșit. Testarea repetată a condiției consumă din timpul de lucru al procesorului, ceea ce va determina o întârziere în program. Dacă programul necesită o întârziere mai mare, puteți mări valoarea de sfârșit.

```
for (contor = 1; contor <= 10000; contor ++)
    ; // Nu face nimic
```

Utilizând astfel de tehnici de întârziere, pot apărea totuși probleme. În primul rând, dacă programul este rulat pe un calculator mai vechi (286, 386 sau 486), lungimea întârzierii va depinde de viteza procesoarelor respective. În al doilea rând, dacă programul este executat într-un mediu multitasking, cum ar fi Windows, OS/2 sau Unix, buclele care „nu fac nimic" consumă timp pe care procesorul îl poate utiliza cu mai mult folos, pentru un alt program. Dacă programele dumneavoastră au nevoie de o astfel de întârziere, consultați funcțiile prezentate în secțiunea „Data și ora".

BUCLA INFINITĂ

C/C++ 120

Așa cum ați învățat, bucla *for* vă permite să repetați una sau mai multe instrucțiuni de un anumit număr de ori. Atunci când bucla *for* îndeplinește condiția sa de sfârșit, programul dumneavoastră își va continua execuția cu instrucțiunea care urmează imediat. Atunci când utilizați bucle *for*, trebuie să vă asigurați că bucla va îndeplini condiția de sfârșit. Altfel, bucla va continua să se execute fără oprire. Astfel de bucle fără sfârșit se numesc *bucle infinite*. În multe cazuri, *buclele infinite* apar ca urmare a unor erori de programare. De exemplu, să considerăm următoarea buclă:

```
for (i = 0; i < 100; i++)
  {
    printf("%d ", i);
    rezultat = valoare * --i; // Cauza erorii
  }
```

Așa cum puteți vedea, cea de a doua instrucțiune a buclei decrementează valoarea variabilei de control *i*. Mai precis, bucla descrește valoarea la –1, iar apoi o crește din nou la 0. Ca urmare, valoarea nu poate ajunge la 100 și bucla nu se va termina niciodată. Atunci când programul dumneavoastră intră într-o buclă infinită, trebuie să apăsați tastele CTRL+C pentru a termina programul. Următorul program, *infinit.c*, conține o buclă infinită:

```
#include <stdio.h>

void main(void)
  {
    int i;
    int rezultat = 0;
    int val = 1;
```

```
    for (i = 0; i < 100; i++)
      {
        printf("%d ", i);
        rezultat = val * --i;
      }
    printf("Rezultat %d\n", rezultat);
 }
```

Atunci când compilați și executați programul *infinit.c*, el va afișa în mod repetat valoarea 0. Pentru a termina programul, apăsați tastele CTRL+C.

121 UTILIZAREA VIRGULEI ÎNTR-O BUCLĂ FOR

C/C++

Așa cum ați învățat, atunci când declarați variabile, limbajul C vă permite să declarați mai multe variabile de același tip, separându-le prin virgulă:

```
int varsta, inaltime, greutate;
```

În plus, limbajul C vă permite să separați cu virgulă inițializările variabilelor:

```
int varsta = 25, inaltime = 173, greutate = 70;
```

În mod similar, limbajul C vă permite să inițializați și să incrementați mai multe variabile într-o buclă *for*, separând operațiile tot prin virgulă. Să considerăm următoarea buclă, în care sunt utilizate variabilele *i* și *j*:

```
for (i = 0, j = 100; i <= 100; i++, j++)
   printf("i = %d j = %d\n", i, j);
```

De cele mai multe ori, veți utiliza buclele *for* cu variabile multiple (cunoscute și sub numele de *bucle imbricate*) în programele care lucrează cu matrice. Veți afla mai multe despre matrice în capitolul „Matrice, pointeri și structuri". Următorul program, *for_2var.c*, ilustrează utilizarea operatorului virgulă într-o buclă *for*:

```
#include <stdio.h>
void main(void)
   {
     int i, j;

     for (i = 0, j = 100; i <= 100; i++, j++)
       printf("i = %d j = %d\n", i, j);
   }
```

122 MODIFICAREA VALORII VARIABILEI DE CONTROL ÎNTR-O BUCLĂ FOR

C/C++

Așa cum ați învățat, bucla *for* vă permite să repetați una sau mai multe instrucțiuni de un anumit număr de ori. Pentru a efectua o astfel de prelucrare, bucla *for* utilizează o *variabilă de control* care lucrează ca un contor. Ca regulă, n-ar trebui să modificați valoarea variabilei

de control în corpul buclei. Singurele locuri în care valoarea variabilei de control poate fi modificată sunt secțiunile de inițializare și de incrementare a buclei. Atunci când modificați valoarea variabilei de control în cadrul instrucțiunilor buclei, vă asumați riscul de a provoca o buclă infinită, iar programul dumneavoastră va deveni mai greu de înțeles.

Totuși, uneori veți dori să terminați bucla sau să săriți peste iterația curentă atunci când variabila de *control* ajunge la o anumită valoare. În asemenea cazuri, utilizați instrucțiunile *break* sau *continue*, care vor fi abordate în secțiunile următoare.

REPETAREA UNEIA SAU MAI MULTOR INSTRUCȚIUNI FOLOSIND BUCLA WHILE

C/C++ 123

Așa cum ați învățat, instrucțiunea *for* vă permite să repetați una sau mai multe instrucțiuni de un anumit număr de ori. Există însă și situații în care programele dumneavoastră trebuie să repete una sau mai multe instrucțiuni până când bucla îndeplinește o condiție care nu implică neapărat contorizarea. De exemplu, dacă scrieți un program care va afișa conținutul unui fișier pe ecran, veți dori ca programul să afișeze fiecare linie a fișierului. De cele mai multe ori, nu veți ști dinainte câte linii conține fișierul. Ca urmare, nu puteți utiliza o buclă *for* pentru a afișa, de exemplu, 100 de linii, deoarece fișierul poate conține mai multe sau mai puține linii, iar dumneavoastră doriți ca programul să citească și să afișeze linii până când întâlnește sfârșitul fișierului. Pentru a realiza aceasta, programul va utiliza bucla *while*. Formatul instrucțiunii *while* este următorul:

```
while (conditie)
   instructiune;
```

Când apare o buclă *while* în program, va fi testată condiția specificată. Dacă este adevărată, se vor efectua instrucțiunile conținute în buclă. Dacă este falsă, execuția programului va continua cu prima instrucțiune care urmează buclei *while*. Instrucțiunea *while* poate să repete o instrucțiune simplă sau o instrucțiune compusă, inclusă între acolade, ca mai jos:

```
while (conditie)
   {
     // Instructiuni
   }
```

Următorul program, *da_nu.c*, utilizează o buclă *while* pentru a cicla până când veți răspunde la o întrebare apăsând tastele D (pentru DA) sau N (pentru NU):

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main(void)
   {
     char litera;     // Litera introdusa de utilizator

     printf("Doriti sa continuati? (D/N): ");

     litera = getch(); // Citeste litera
     litera = toupper(litera); // Converteste in majuscula

     while ((litera != 'D') && (litera != 'N'))
```

```
    {
      putch(7); // Emite un sunet
      litera = getch(); // Citeste litera
      litera = toupper(litera); // Converteste in majuscula
    }
  printf("\n Raspunsul a fost %c\n", litera);
}
```

Mai întâi programul va afişa mesajul pe care îl conţine prima instrucţiune *printf*. Apoi programul va utiliza funcţia *getch* pentru a citi tasta apăsată. Pentru a simplifica testarea, programul va converti litera în majusculă, astfel încât bucla să testeze numai pentru N sau D. În al treilea rând, bucla *while* va testa litera introdusă de utilizator. Dacă aceasta este D sau N, condiţia nu va fi realizată, iar instrucţiunile buclei nu se vor executa. Dacă litera introdusă nu este D sau N, condiţia buclei este adevărată şi instrucţiunile sale se vor executa. În cadrul buclei, programul va emite un sunet (bip) prin difuzorul încorporat, pentru a indica un caracter nevalabil. Apoi programul va citi noua tastă apăsată şi va converti în majusculă caracterul introdus. Bucla va repeta apoi testul pentru a determina dacă utilizatorul a tastat un D sau un N. Dacă a fost introdus alt caracter, instrucţiunile buclei se vor repeta. Altfel, execuţia programului va continua cu prima instrucţiune care urmează buclei.

124 PĂRŢILE UNEI BUCLE WHILE



O buclă *while* vă permite să executaţi una sau mai multe comenzi până când programul îndeplineşte condiţiile specificate. În secţiunea 114, aţi învăţat că o buclă *for* conţine de obicei patru secţiuni: o iniţializare, un test, o instrucţiune şi un increment. Spre deosebire de aceasta, bucla *while* conţine numai un test şi instrucţiunile care se repetă, aşa cum se vede mai jos:

```
while (conditie)
   instructiune;
```

Aşa cum aţi învăţat în secţiunea 120, o buclă infinită este o buclă a cărei condiţie de sfârşit nu este îndeplinită niciodată şi, ca urmare, execuţia continuă la nesfârşit. Atunci când scrieţi programe care utilizează bucle *while*, puteţi reduce posibilitatea de a produce bucle infinite asigurându-vă că bucla *while* efectuează aceiaşi patru paşi pe care îi efectuează o buclă *for*. Pentru a vă reaminti mai uşor cei patru paşi, puteţi reţine acronimul *ITEM*, a cărui semnificaţie este prezentată în tabelul 124:

Acţiune	Descriere
Iniţializare	Iniţializează variabila de control a buclei
Test	Testează variabila de control a buclei sau condiţia
Execuţie	Execută instrucţiunile din corpul buclei
Modificare	Modifică valoarea variabilei de control sau efectuează o operaţie care afectează condiţia pe care o testaţi.

***Tabelul 124** Componentele acronimului **ITEM**.*

Spre deosebire de bucla *for*, care vă permite să iniţializaţi şi să incrementaţi în mod explicit variabila de control, o buclă *while* impune includerea în program a unor instrucţiuni care să

execute aceste acțiuni. Următorul program, *item.c*, ilustrează modul în care program efectuează acești pași. Spre deosebire de programele pe care le-ați scris până acum, *iten* utilizează o buclă *while* pentru a afișa numerele de la 1 la 100:

```
#include <stdio.h>

void main(void)
  {
    int contor = 1;          // Initializeaza variabila de contro
    while (contor <= 100) // Testeaza variabila de control
      {
        printf("%d ", contor); // Executa instructiunile
        contor++;            // Modifica variabila de control
      }
  }
```

Dacă scrieți un program care utilizează o buclă *while* și acesta produce o buclă infinit înseamnă că una dintre operațiile ITEM nu este corectă.

REPETAREA UNEIA SAU MAI MULTOR INSTRUCȚIUNI UTILIZÂND DO

C/C++ 125

Așa cum ați învățat, instrucțiunea *while* vă permite să repetați una sau mai multe instrucțiu până când este îndeplinită o condiție dată. În mod asemănător, instrucțiunea *for* vă permi să repetați una sau mai multe instrucțiuni de un anumit număr de ori. În plus, limbajul oferă și instrucțiunea *do*, care execută una sau mai multe instrucțiuni cel puțin o dată și apo dacă este necesar, repetă instrucțiunile. Formatul instrucțiunii *do* este următorul:

```
do
   instructiune;
while (conditie);
```

Instrucțiunea *do* este ideală pentru situațiile care vă cer să efectuați una sau mai mul instrucțiuni cel puțin o dată. Să considerăm, de exemplu, următorul fragment de cod:

```
printf("Doriti sa continuati? (D/N): ");
litera = getch();    // Citeste litera
litera = toupper(litera); // Converteste in majuscula
while ((litera != 'D') && (litera != 'N'))
  {
    putch(7);              // Emite un sunet
    litera = getch(); // Citeste litera
    litera = toupper(litera); // Converteste in majuscula
  }
```

Așa cum puteți vedea, programul cere utilizatorul să apese o tastă, apoi citește caracteru introdus și îl convertește în majusculă. În funcție de tasta apăsată de utilizator, programul v: începe o buclă *while* care efectuează aceleași comenzi. Observați în exemplul următor cum puteți simplifica programul utilizând instrucțiunea *do*:

```
printf("Doriti sa continuati? (D/N): ");
do
  {
    litera = getch();          // Citeste litera
    litera = toupper(litera); // Converteste in majuscula
    if ((litera != 'D') && (litera != 'N'))
      putch(7); // Emite un sunet semnaland litera nevalabila
  }
while ((litera != 'D') && (litera != 'N'))
```

Atunci când apare o instrucţiune *do* în program, se vor executa intrucţiunile dintre cuvintele *do* şi *while*, apoi se testează condiţia pe care o specifică clauza *while* pentru a determina dacă instrucţiunile se mai repetă sau nu. Prin urmare, instrucţiunile dintr-o buclă *do* se vor executa întotdeauna cel puţin o dată. Programele utilizează frecvent bucla *do* pentru afişarea şi prelucrarea opţiunilor unui meniu. Următorul program, *do_meniu.c*, utilizează instrucţiunea *do* pentru afişarea şi prelucrarea opţiunilor unui meniu până când utilizatorul selectează opţiunea Iesire:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>

void main(void)
  {
    char litera;

    do
      {
        printf("A Afiseaza continutul directorului\n");
        printf("B Modifica ora sistemului\n");
        printf("C Modifica data sistemului\n");
        printf("E Iesire\n");
        printf("Alegerea dumneavoastra: ");
        litera = getch();
        litera = toupper(litera);
        if (litera == 'A')
          system("DIR");
        else if (litera == 'B')
          system("TIME");
        else if (litera == 'C')
          system("DATE");
      }
    while (litera != 'E');
  }
```

INSTRUCȚIUNEA CONTINUE

C/C++ 126

Așa cum ați învățat, instrucțiunile *for*, *while* și *do* permit programelor dumneavoastră să repete una sau mai multe instrucțiuni de un anumit număr de ori până când o anumită condiție este evaluată ca adevărată sau falsă. Uneori vor apărea situații în care veți dori să fie sărită iterația curentă în funcție de o a doua condiție. Instrucțiunea *continue* a limbajului C vă permite să faceți exact acest lucru. Dacă într-o buclă *for* apare instrucțiunea *continue*, se va executa imediat secțiunea de incrementare a buclei și apoi se va testa condiția de sfârșit. Dacă această instrucțiune apare într-o structură *do* sau *while*, atunci se va efectua imediat testarea condiției de final. Pentru a înțelege mai bine, să considerăm următorul program, *par_imp.c*, care utilizează instrucțiunea *continue* într-o buclă *for* și într-una *while* pentru a afișa valorile pare și impare dintre 1 și 100:

```
#include <stdio.h>
void main(void)
  {
    int contor;

    printf("\nValori pare\n");
    for (contor = 1; contor <= 100; contor++)
    {
      if (contor % 2) // Impar
      continue;
      printf("%d ", contor);
    }
    printf("\nValori impare\n");
    contor = 0;
    while (contor <= 100)
    {
      contor++;
      if (! (contor % 2)) // Par
      continue;
      printf("%d ", contor);
    }
  }
```

Programul utilizează operatorul *modulo* (rest) pentru a determina dacă o valoare este pară sau impară. Dacă împărțiți o valoare cu 2 și obțineți restul 1, valoarea este impară. Dacă însă obțineți restul 0, valoarea este pară.

Este important să observați că, în mod normal, puteți elimina instrucțiunea *continue* rescriindu-vă instrucțiunile *if* și *else* din program. De exemplu, următorul program, *nu_cont.c*, afișează de asemenea numerele pare și impare, fără să utilizeze instrucțiunea *continue*.

```
#include <stdio.h>
void main(void)
  {
    int contor;

    printf("\nValori pare\n");
```

```
    for (contor = 1; contor <= 100; contor++)
      {
        if (!(contor % 2)) // Par
        printf("%d ", contor);
      }
    printf("\nValori impare\n");
    contor = 0;
    while (contor <= 100)
      {
        contor++;
        if (contor % 2) // Impar
        printf("%d ", contor);
      }
  }
```

Înainte de a plasa o instrucţiune *continue* în program, examinaţi cu atenţie codul pentru a vedea dacă nu cumva puteţi scrie aceleaşi instrucţiuni fără a utiliza *continue*. În multe cazuri, veţi găsi că este mai uşor de înţeles codul fără instrucţiunea *continue*.

127 ÎNCHEIEREA BUCLEI FOLOSIND INSTRUCŢIUNEA BREAK



Aşa cum aţi învăţat, instrucţiunile *for*, *while* şi *do* permit programului să repete una sau mai multe instrucţiuni până când o condiţie specificată este evaluată ca adevărată sau falsă. Uneori vor apărea situaţii în care veţi dori ca programul să întrerupă imediat bucla curentă în funcţie de o a doua condiţie şi să continue prelucrarea de la următoarea instrucţiune care urmează buclei. Instrucţiune *break* a limbajului C vă permite să faceţi exact acest lucru. Când apare o instrucţiune *break* în cadrul unei bucle, executarea acestei bucle se încheie imediat. Următoarea instrucţiune pe care programul o va executa este cea care urmează imediat după buclă. În cazul unei bucle *for*, nu se va mai executa secţiunea de incrementare a buclei, ci se va opri imediat. Următorul program, *b_break.c*, ilustrează utilizarea instrucţiunii *break*. Programul numără de la 1 la 100 şi apoi de la 100 la 1. De fiecare dată când bucla ajunge la 50, instrucţiunea *break* face ca execuţia buclei să se oprească:

```
#include <stdio.h>
void main(void)
  {
    int contor;
    for (contor = 1; contor <= 100; contor++)
      {
        if (contor == 50)
        break;
        printf("%d ", contor);
      }
    printf("\nBucla urmatoare\n");
    for (contor = 100; contor >= 1; contor--)
```

```
    {
      if (contor == 50)
      break;
      printf("%d ", contor);
    }
  }
```

Ca și în cazul instrucțiunii *continue*, puteți de obicei să rescrieți condițiile de ciclare și structurile *if-else* ale programului dumneavoastră pentru a elimina necesitatea instrucțiunii *break* în cadrul buclelor. De multe ori, după ce vă rescrieți programul eliminând instrucțiunea *break*, acesta va deveni mult mai ușor de înțeles pentru cititor. Ca regulă, folosiți instrucțiunea *break* numai în cadrul instrucțiunii *switch*.

RAMIFICAREA CU AJUTORUL INSTRUCȚIUNII GOTO C/C++ 128

Dacă ați programat în BASIC, FORTRAN sau în limbaj de asamblare, probabil că v-ați obișnuit să implementați operațiile *if-else* sau buclele utilizând instrucțiunea GOTO. Ca majoritatea limbajelor de programare, C oferă și o instrucțiune *goto*, care permite ca execuția programului să se ramifice trecând la o anumită locație, denumită *etichetă*. Formatul instrucțiunii *goto* este următorul:

```
goto eticheta;
eticheta:
```

Următorul program, *goto_100.c*, utilizează instrucțiunea *goto* pentru a afișa numerele de la 1 la 100:

```
#include <stdio.h>
void main(void)
  {
    int nr = 1;
    eticheta:
      printf("%d ", nr++);

      if (nr <= 100)
      goto eticheta;
  }
```

Atunci când utilizați instrucțiunea *goto*, eticheta trebuie să se afle în funcția curentă. Cu alte cuvinte, nu puteți utiliza *goto* pentru ramificarea de la *main* la o etichetă care apare într-o altă funcție sau invers.

Ținând cont că, de multe ori, programatorii au utilizat eronat această instrucțiune în trecut, ar trebui să o evitați de câte ori este posibil și să folosiți, în schimb, structuri ca *if*, *if-else* și *while*. De cele mai multe ori, puteți utiliza aceste trei construcții pentru a rescrie un fragment de cod care folosește *goto*, obținând un program mult mai ușor de citit.

129 TESTAREA CONDIȚIILOR MULTIPLE



Așa cum ați învățat, instrucțiunea *if-else* din limbajul C vă permite să testați condiții multiple. De exemplu, să considerăm următoarea testare a variabilei *litera*:

```
litera = getch();
litera = toupper(litera);

if (litera == 'A')
   system("DIR");
else if (litera == 'B')
   system("TIME");
else if (litera == 'C')
   system("DATE");
```

Pentru cazul în care testați aceeași variabilă pentru mai multe valori posibile, limbajul C oferă instrucțiunea *switch*, care are următorul format:

```
switch (expresie) {
   case Constanta_1: instructiune;
   case Constanta_2: instructiune;
   case Constanta_3: instructiune;
     :  :  :             :
   };
```

În loc să utilizați instrucțiunile *if-else* anterioare, puteți folosi instrucțiunea *switch*, așa cum se vede mai jos:

```
switch (litera) {
   case 'A': system("DIR");
             break;
   case 'B': system("TIME");
             break;
   case 'C': system("DATE");
             break;
   };
```

Atunci când într-un program apare o instrucțiune *switch*, se evaluează expresia care urmează și apoi se compară rezultatul cu fiecare dintre valorile constante specificate după cuvântul cheie *case*. Dacă una dintre aceste valori se potrivește, se vor executa instrucțiunile corespunzătoare. Instrucțiunea *break* separă instrucțiunile asociate fiecărui caz. De obicei, veți plasa o instrucțiune *break* după ultima instrucțiune care corespunde unei opțiuni. În secțiunea 130, veți învăța detaliile legate de utilizarea instrucțiunii *break* într-o structură *switch*. Următorul program, *swt_men.c*, utilizează instrucțiunea *switch* pentru a prelucra selectarea de către utilizator a opțiunilor dintr-un meniu:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
void main(void)
  {
    char litera;

    do {
      printf("A Afiseaza continutul directorului\n");
      printf("B Modifica ora sistemului\n");
      printf("C Modifica data sistemului\n");
      printf("E Iesire\n");
      printf("Alegerea dumneavoastra: ");
      litera = getch();
      litera = toupper(litera);
      switch (litera) {
         case 'A': system("DIR");
                   break;
         case 'B': system("TIME");
                   break;
         case 'C': system("DATE");
                   break;
      };
    }
    while (litera != 'E');
  }
```

INSTRUCȚIUNEA BREAK DIN SWITCH

C/C++ 130

În secțiunea 129, ați învățat că instrucțiunea *switch* vă permite să efectuați o procesare condițională. Așa cum ați învățat, puteți specifica unul sau mai multe cazuri posibile utilizând această instrucțiune. Pentru fiecare caz, se specifică instrucțiunile corespunzătoare și, în mod normal, se plasează o instrucțiune *break* la sfârșitul lor, pentru a separa o instrucțiune *case* de alta. Dacă omiteți instrucțiunea *break*, execuția va continua cu instrucțiunile care urmează, fără să se țină seama de cazul căruia îi sunt asociate instrucțiunile. De exemplu, să considerăm următoarea instrucțiune *switch*:

```
switch (litera) {
   case 'A': system("DIR");
   case 'B': system("TIME");
   case 'C': system("DATE");
};
```

Dacă variabila *litera* conține litera A, deci corespunde primului caz, se va executa comanda DIR. Totuși, deoarece nu urmează nici o instrucțiune *break*, programul va executa de asemenea și comenzile TIME și DATE. Pentru a preveni executarea instrucțiunilor altui caz, folosiți instrucțiunea *break*, așa cum se vede mai jos:

```
switch (litera) {
    case 'A': system("DIR");
```

```
            break;
    case 'B': system("TIME");
            break;
    case 'C': system("DATE");
            break;
   };
```

Uneori, veți dori ca programul dumneavoastră să execute în cascadă mai multe instrucțiuni *case*. De exemplu, următorul program, *vocale.c*, utilizează o instrucțiune *switch* pentru a număra vocalele din alfabet:

```
#include <stdio.h>

void main(void)
  {
    char litera;
    int nr_vocale = 0;

    for (litera = 'A'; litera <= 'Z'; litera++)
    switch (litera) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': nr_vocale++;
      };
    printf("Numarul de vocale este %d\n", nr_vocale);
  }
```

În acest caz, dacă variabila *litera* conține unul dintre caracterele A, E, I sau O, s-a realizat unul dintre cazuri, iar compilatorul va „coborî" până la instrucțiunea corespunzătoare literei U, care incrementează variabila *nr_vocale*. Deoarece instrucțiunea *switch* nu conține alte cazuri după litera U, programul nu conține instrucțiunea *break*.

131 *UTILIZAREA CAZULUI DEFAULT AL INSTRUCȚIUNII SWITCH*



Așa cum ați învățat, instrucțiunea *switch* vă permite să efectuați o procesare condițională. Atunci când utilizați această instrucțiune, specificați unul sau mai multe cazuri, ca mai jos:

```
switch (litera) {
    case 'A': system("DIR");
            break;
    case 'B': system("TIME");
            break;
    case 'C': system("DATE");
            break;
   };
```

Când utilizați instrucțiunea *switch*, pot apărea situații în care doriți ca programul să efectueze anumite instrucțiuni atunci când celelalte cazuri nu se potrivesc. Pentru aceasta, puteți include cazul *default* (prestabilit) în instrucțiunea *switch*, ca mai jos:

```
switch (expresie) {
  case Constanta_1: instructiune;
  case Constanta_2: instructiune;
  case Constanta_3: instructiune;
   :   :          :
  default: instructiune;
  };
```

Dacă nu se potrivește nici una dintre opțiunile care preced cazul *default*, se vor executa instrucțiunile corespunzătoare acestuia. Următorul program, *con_voc.c*, utilizează cazul *default* pentru a număra consoanele din alfabet:

```
#include <stdio.h>

void main(void)
  {
    char litera;
    int nr_vocale = 0;
    int nr_consoane = 0;

    for (litera = 'A'; litera <= 'Z'; litera++)
    switch (litera) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': nr_vocale++;
      break;
      default: nr_consoane++;
    };

    printf("Numarul de vocale este %d\n", nr_vocale);
    printf("Numarul de consoane este %d\n", nr_consoane);
  }
```

DEFINIREA CONSTANTELOR

C/C++ 132

Ca regulă, puteți îmbunătăți lizibilitatea și portabilitatea programului înlocuind referirile la numere, cum ar fi 512, cu constante sugestive. O *constantă* este un nume pe care compilatorul de C îl asociază unei valori care nu se modifică. Pentru a crea o constantă, se utilizează directiva *#define*. De exemplu, următoarea directivă creează o constantă denumită *DIM_LINIE* și îi atribuie valoarea 128:

```
#define DIM_LINIE 128
```

Atunci când preprocesorul de C va întâlni numele constantei *DIM_LINIE* în program, îl va înlocui cu valoarea ei. De exemplu, să considerăm următoarea declarație de șiruri:

```
char linie[128];
char text[128];

char linie_curenta[DIM_LINIE];
char sir_introdus[DIM_LINIE];
```

Primele două declarații creează șiruri de caractere care conțin 128 de biți fiecare. Următoarele două declarații creează șiruri de caractere de dimensiunea constantei *DIM_LINIE*. Atunci când alți programatori vă citesc programul, una dintre primele întrebări pe care le pun este de ce ați pus tocmai 128 în declarația șirului. În cazul celei de a doua declarații, programatorul știe însă că ați declarat toate șirurile în funcție de o dimensiune predefinită: *DIM_LINIE*. În cadrul programelor, puteți include bucle, la fel ca în exemplele următoare:

```
for (i = 0; i < 128; i++)
   // Instructiuni
for (i = 0; i < DIM_LINIE; i++)
   // Instructiuni
```

Cea de a doua buclă *for* face programele dumneavoastră mult mai ușor de citit și de modificat. Presupunând, de exemplu, că programul utilizează peste tot valoarea 128 pentru a se referi la dimensiunea șirului, dacă mai târziu doriți să schimbați dimensiunea la 256 de caractere, va trebui să modificați fiecare apariție a valorii 128 în program, ceea ce durează mult. Dacă însă utilizați o constantă cum ar fi *DIM_LINIE*, va fi nevoie să modificați numai directiva *#define* – un proces cu un singur pas, așa cum se vede mai jos:

```
#define DIM_LINIE 256
```

133 EXTINDEREA MACROINSTRUCȚIUNILOR ȘI A CONSTANTELOR

C/C++

În secțiunea 132, ați învățat că puteți utiliza directiva *#define* pentru a defini o constantă în cadrul programului dumneavoastră. De exemplu, următorul program, *cst_macr.c*, utilizează trei constante:

```
#define LINIE 128
#define TITLU "Totul despre C/C++"
#define CAPITOL "Macroinstructiuni"

void main(void)
  {
    char carte[LINIE];
    char nume[LINIE];

    printf("Titlul cartii este %s\n", TITLU);
    printf(CAPITOL);
  }
```

Atunci când compilați un program în C, mai întâi se rulează un program denumit *preprocesor*. Scopul preprocesorului este de a include fișierele antet specificate și de a

extinde macroinstrucțiunile și constantele. Înainte de a începe compilarea, prepocesorul va înlocui fiecare nume de constantă cu valoarea ei, ca mai jos:

```
void main(void)
  {
    char carte[128];
    char nume[128];

    printf("Titlul cartii este %s\n", "Totul despre C/C++");
    printf("Macroinstructiunile");
  }
```

Deoarece preprocesorul lucrează cu *#include*, *#define* și alte instrucțiuni introduse cu caracterul #, acestea sunt adesea denumite *directive către preprocesor*.

ATRIBUIREA DE NUME CONSTANTELOR ȘI MACRONSTRUCȚIUNILOR

C/C++ 134

După cum ați învățat, o constantă este un nume pe care compilatorul de C îl asociază unei valori care nu se modifică. În secțiunea 144, veți învăța despre macroinstrucțiunile limbajului C. Atunci când folosiți macroinstrucțiuni sau constante într-un program, trebuie să le dați nume sugestive, care descriu cu acuratețe scopul lor. Pentru a ajuta programatorii care vă citesc codul să facă diferența între constante și variabile, trebuie să scrieți cu majuscule numele constantelor și ale macroinstrucțiunilor. Iată câteva exemple de macroinstrucțiuni:

```
#define TRUE 1
#define FALSE 0
#define PI 3.1415
#define PROGRAMATOR "Kris Jamsa"
```

Așa după cum puteți vedea, constantele pot conține valori de tipul *int*, *float* sau chiar *char*.

UTILIZAREA CONSTANTEI PREDEFINITE __FILE__

C/C++ 135

Atunci când veți lucra la un proiect mare, veți dori uneori ca preprocesorul să cunoască numele fișierului sursă curent. De exemplu, puteți utiliza numele fișierului în cadrul unei directive către preprocesor care include un mesaj către utilizator spunând că programul este încă în lucru, ca mai jos:

```
Programul PLATI.C este in stadiu de dezvoltare si testare.
Aceasta este doar o versiune BETA.
```

Pentru a ajuta programul să efectueze astfel de procesări, preprocesorul de C definește constanta *__FILE__* ca fiind egală cu numele fișierului sursă curent. Următorul program, *filecons.c*, ilustrează modul de utilizare a constantei *__FILE__*:

```
#include <stdio.h>

void main(void)
  {
    printf("Fisierul %s este in stadiu de test\n", __FILE__);
  }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră vor apărea următoarele:

```
Fisierul filecons.c este in stadiu de test
C:\>
```

Observație: *Spre deosebire de alte constante ale preprocesorului, care se schimbă de la un compilator la compilator, constanta* ***__FILE__*** *este definită la fel în cadrul compilatoarelor* ***Turbo C++ Lite, Visual C++, Borland C++ 5.02*** *și* ***Borland C++ Builder.***

136 UTILIZAREA CONSTANTEI PREDEFINITE __LINE__ C/C++

Atunci când veți lucra la un proiect mare, veți dori uneori ca preprocesorul să cunoască și eventual să utilizeze numărul liniei curente al fișierului sursă. De exemplu, dacă depanați un program, veți dori să fie afișate mesaje pe măsură ce compilatorul trece prin diverse puncte ale programului, ca mai jos:

```
Am trecut de linia 10
Am trecut de linia 301
Am trecut de linia 213
```

Următorul program, *linecons.c*, ilustrează modul de utilizare a constantei *__LINE__*:

```
#include <stdio.h>

void main(void)
  {
    printf("Am trecut de linia %d\n", __LINE__);
    // Alte instructiuni
    printf("Am trecut de linia %d\n", __LINE__);
  }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră vor apărea următoarele:

```
Am trecut de linia 5
Am trecut de linia 7
C:\>
```

Observație: *Spre deosebire de alte constante ale preprocesorului, care se schimbă de la un compilator la compilator, constanta* ***__LINE__*** *este definită la fel în cadrul compilatoarelor* ***Turbo C++ Lite, Visual C++, Borland C++ 5.02*** *și* ***Borland C++ Builder.***

137 MODIFICAREA NUMĂRULUI CURENT AL LINIEI C/C++

În secțiunea 136, ați învățat cum să utilizați constanta *__LINE__* în programele dumneavoastră. Atunci când utilizați această constantă, s-ar putea ca uneori să doriți să schimbați numărul curent al liniei preprocesorului. De exemplu, să presupunem că utilizați constanta *__LINE__* pentru a vă ajuta să depanați programul, așa cum am arătat în secțiunea 136. Dacă ați stabilit că eroarea se află într-un anumit set de instrucțiuni, probabil că veți dori ca preprocesorul să afișeze numărul liniei pornind de la o anumită locație. Pentru a vă ajuta să realizați aceasta, preprocesorul de C oferă directiva *#line*, care vă permite să schimbați numărul de ordine al liniei curente. Următoarea directivă, de exemplu, indică preprocesorului să stabilească numărul de ordine al liniei curente la 100:

```
#line 100
```

De asemenea, puteţi utiliza această directivă pentru a schimba numele fişierului sursă pe care îl va afişa constanta *__FILE__*:

```
#line 1 "NUMEFIS.C"
```

Următorul program, *sch_line.c*, ilustrează modul de utilizare a directivei *#line*:

```
#include <stdio.h>
void main(void)
  {
    printf("Fisier %s: Am trecut de linia %d\n", __FILE__,
      __LINE__);
    // Alte instructiuni
  #line 100 "NUMEFIS.C"
    printf("Fisier %s: Am trecut de linia %d\n", __FILE__,
      __LINE__);
  }
```

Atunci când compilaţi şi executaţi acest program, pe ecranul dumneavoastră vor apărea următoarele:

```
Fisier sch_line.c: Am trecut de linia 5
Fisier NUMEFIS.C: Am trecut de linia 101
C:\>
```

GENERAREA UNEI ERORI NECONDIŢIONATE

C/C++ 138

Pe măsură ce programele dumneavoastră vor deveni complexe şi vor utiliza un mare număr de fişiere antet, vor apărea situaţii în care veţi dori ca programul să nu se compileze cu succes dacă nu au fost definite una sau mai multe constante. Tot aşa, dacă lucraţi cu un grup de programatori şi doriţi să-i avertizaţi asupra modificărilor pe care le-aţi făcut în program, puteţi utiliza directiva către preprocesor *#error* pentru a afişa un mesaj de eroare şi a încheia compilarea. Următoarea directivă, de exemplu, încheie compilarea afişând un mesaj către utilizator în legătură cu actualizarea realizată:

```
#error Procedura sort_sir utilizeaza siruri far
```

Înainte ca alţi programatori să poată compila cu succes programul, ei trebuie să înlăture directiva *#error*, luând astfel cunoştinţă despre modificare.

ALTE CONSTANTE DE PREPROCESOR

C/C++ 139

Unele secţiuni din acest capitol au prezentat constante de preprocesor pe care le acceptă majoritatea compilatoarelor. Unele compilatoare definesc multe alte constante de acest fel. Compilatorul Microsoft *Visual C++*, de exemplu, utilizează încă 15 constante de preprocesor, pe care această carte nu le discută. Consultaţi documentaţia care însoţeşte compilatorul dumneavoastră pentru a stabili dacă programele pot beneficia de alte constante similare. În plus, puteţi consulta documentaţia on-line, capitolul *Predefined Macros*.

140 ÎNREGISTRAREA DATEI ȘI OREI PREPROCESORULUI



Când lucrați la programe extinse, probabil că veți dori ca preprocesorul dumneavoastră să lucreze cu data și ora curente. De exemplu, probabil că veți dori ca programul să afișeze un mesaj conținând data și ora ultimei compilări, ca mai jos:

```
Versiune Beta: plati.c Ultima compilare Nov 4 1997 12:00:00
```

Pentru a vă ajuta să faceți aceasta, preprocesorul de C atribuie constantelor *__DATE__* și *__TIME__* data și ora curente. Următorul program, *date_time.c*, ilustrează utilizarea acestei constante:

```
#include <stdio.h>
void main(void)
  {
    printf("Versiune Beta: Ultima compilare %s %s\n", __DATE__,
      __TIME__);
  }
```

141 TESTAREA COMPATIBILITĂȚII CU ANSI C



Cu toate că cele mai multe compilatoare de C sunt foarte asemănătoare, fiecare oferă facilități unice. Pentru a vă ajuta să scrieți programe care se pot muta cu ușurință de la un sistem la altul, Institutul Național American pentru Standarde (ANSI) a definit standarde pentru operatorii, structurile, instrucțiunile și funcțiile pe care trebuie să le accepte un compilator. Compilatoarele compatibile cu aceste standarde sunt denumite *compilatoare ANSI C*. Pe măsură ce veți crea diferite programe, uneori veți dori să știți dacă utilizați sau nu un compilator ANSI. Pentru aceasta, compilatoarele ANSI C definesc constanta *__STDC__* (STandarD C). Dacă este definită această constantă, compilatorul este compatibil cu standardul ANSI. Dacă nu este definită, compilatorul nu este compatibil. Următorul program, *tst_ansi.c*, utilizează constanta *__STDC__* pentru a testa compatibilitatea compilatorului curent cu standardul ANSI.

```
#include <stdio.h>

void main(void)
  {
   #ifdef __STDC__
     printf("Compatibilitate ANSI C\n");
   #else
     printf("Nu este in modul ANSI C\n");
   #endif
  }
```

***Observație:** Cele mai multe compilatoare acceptă comutatoare în linia de comandă sau directive pragma in-line care indică utilizarea standardului ANSI. Opțiunile din linia de comandă și directivele pragma vor fi prezentate mai târziu.*

TESTAREA MODULUI DE LUCRU AL COMPILATORULUI (C++ SAU C)

C/C++ 142

Unele dintre sugestiile prezentate în această carte sunt valabile atât pentru programarea în C, cât și pentru programarea în C++, pe când altele se aplică numai pentru C++. Când vă veți crea propriile programe, uneori veți dori ca preprocesorul să determine dacă utilizați un compilator de C sau de C++ și să proceseze instrucțiunile dumneavoastră în mod corespunzător. Pentru a realiza aceasta, multe compilatoare de C++ definesc constanta *__cplusplus*. Dacă utilizați un compilator de C standard, constanta va fi nedefinită. Următorul program, *test_cpp.c*, utilizează constanta *__cplusplus* pentru a determina modul de lucru curent al compilatorului:

```
#include <stdio.h>

void main(void)
  {
    #ifdef __cplusplus
      printf("Se utilizeaza C++\n");
    #else
      printf("Se utilizeaza C\n");
    #endif
  }
```

Dacă examinați fișierele antet oferite de compilator, veți întâlni foarte des această constantă.

***Observație:** Cele mai multe compilatoare acceptă opțiuni în linia de comandă care le indică să compileze folosind C++ sau limbajul C standard.*

ELIMINAREA DEFINIȚIEI UNEI MACROINSTRUCȚIUNI SAU A UNEI CONSTANTE

C/C++ 143

Câteva secțiuni din acest capitol au analizat constantele și macroinstrucțiunile definite de preprocesor sau incluse într-un fișier antet. În funcție de programul dumneavoastră, uneori veți dori ca preprocesorul să înlăture sau să modifice definiția uneia sau mai multor constante. De exemplu, următorul fragment de cod modifică macroinstrucțiunea *_toupper*, care este definită în fișierul antet *ctype.h*:

```
#define _toupper(c) ((((c) >= 'a')&&((c) <= 'z')) ? (c) - 'a' +
'A' : c)
```

Atunci când compilați acest program, cele mai multe compilatoare vor afișa un mesaj prin care vă avertizează că ați redefinit această macroinstrucțiune. Pentru a evita afișarea acestui mesaj de avertizare, puteți să înlăturați definiția curentă a macroinstrucțiunii cu directiva *#undef* și apoi să o redefiniți, ca mai jos:

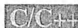
```
#undef _toupper
#define _toupper(c) ((((c) >= 'a')&&((c) <= 'z')) ? (c) - 'a' +
'A' : c)
```

144 COMPARAȚIE ÎNTRE MACROINSTRUCȚIUNI ȘI FUNCȚII



Programatorii în C începători se încurcă atunci când trebuie să aleagă între utilizarea unei macroinstrucțiuni și cea a unei funcții din cauza similarității dintre cele două. După cum ați învățat, de fiecare dată când preprocesorul întâlnește o referire la o macroinstrucțiune într-un program, o înlocuiește cu instrucțiunile componente. Prin urmare, dacă programul dumneavoastră utilizează de 15 ori o anumită macroinstrucțiune, vor fi inserate 15 copii diferite printre celelalte instrucțiuni ale lui. Ca urmare, dimensiunea programului executabil va crește. Pe de altă parte, atunci când programul dumneavoastră utilizează o funcție, el conține numai o copie a codului, ceea ce îi reduce dimensiunea. Când utilizează o funcție, programul apelează codul funcției (se ramifică spre el). Totuși, funcțiile au dezavantajul prelucrărilor suplimentare pe care le implică fiecare apel, astfel că execuția funcției durează puțin mai mult decât cea a unei macroinstrucțiuni echivalente. Prin urmare, dacă doriți rapiditate, utilizați o macroinstrucțiune. Dacă vă interesează mai mult dimensiunea programului, utilizați o funcție.

145 DIRECTIVELE PRAGMA



Câteva secțiuni din acest capitol v-au prezentat diferite directive către preprocesor, cum ar fi *#define*, *#include* și *#undef*. În funcție de compilatorul dumneavoastră, preprocesorul poate accepta diferite directive către compilator, denumite *pragma*. Formatul unei directive *pragma* este:

```
#pragma directiva_compilator
```

De exemplu, compilatorul *Turbo C++ Lite* vă pune la dispoziție directivele pragma *startup* și *exit*, care vă permit să specificați funcțiile ce vor fi executate automat atunci când începe sau se termină programul.

```
#pragma startup incarca_date
#pragma exit inchide_fisiere
```

Observați că funcția pe care o numiți în cadrul directivei pragma *startup* se va executa *înainte de main*, deci nu ar trebui să o utilizați prea des. În funcție de compilatorul dumneavoastră, veți avea la dispoziție diferite directive pragma. Consultați documentația care însoțește compilatorul pentru a afla toate amănuntele referitoare la directivele pragma disponibile.

Observație: *Atunci când utilizați directivele pragma **startup** și **exit**, trebuie ca funcțiile apelate să nu aibă nici un parametru și să nu returneze nici o valoare. Cu alte cuvinte, trebuie să scrieți funcția în următorul mod:*

```
void functie(void)
```

146 VALORILE PREDEFINITE ȘI MACROINSTRUCȚIUNILE



Câteva secțiuni din acest capitol s-au referit la macroinstrucțiuni, constante și diferite directive către preprocesor. Una dintre cele mai eficiente metode de a învăța cum se utilizează macroinstrucțiunile, constantele și alte directive către preprocesor, este examinarea

modului în care compilatorul de C lucrează cu aceste opțiuni. Compilatorul de C plasează macroinstrucțiunile și constantele în fișiere antet din subdirectorul *include* al compilatorului. Multe fișiere antet prezintă diferite modalități de a utiliza directivele către preprocesor. Ar fi bine să examinați conținutul diferitelor fișiere antet, pentru a vedea diferite metode de a îmbunătăți programele folosind avantajele acestor capacități ale preprocesorului.

CREAREA PROPRIILOR FIȘIERE ANTET

C/C++ 147

După cum știți, compilatorul de C oferă diverse fișiere antet care conțin macroinstrucțiuni, constante și prototipuri de funcții. Pe măsură ce veți crea tot mai multe programe, veți observa că multe dintre ele utilizează aceleași constante și macroinstrucțiuni. În loc să scrieți în mod repetat aceleași macroinstrucțiuni și constante, este recomandabil să creați propriul dumneavoastră fișier antet și să le plasați în el. Presupunând că ați creat un fișier denumit *ant_meu.h*, puteți include acest fișier la începutul programului, utilizând directiva de preprocesor *#include*, ca mai jos:

```
#include "ant_meu.h"
```

Atunci când includeți constante și macroinstrucțiuni prin intermediul unui fișier antet, puteți modifica rapid mai multe programe reeditând fișierul antet și apoi recompilând programele care îl includ.

UTILIZAREA DIRECTIVELOR #INCLUDE <NUMEFIS.H> SAU #INCLUDE "NUMEFIS.H"

C/C++ 148

În toate programele prezentate până acum, fișierul antet *stdio.h* era inclus ca mai jos:

```
#include <stdio.h>
```

În secțiunea 147, ați învățat cum să creați și să includeți propriul dumneavoastră fișier antet, *ant_meu.h*. Puteți să includeți atât *stdio.h* cât și *ant_meu.h* în programele dumneavoastră, cu următoarele instrucțiuni:

```
#include "ant_meu.h"
#include <stdio.h>
```

Examinând cele două instrucțiuni *include*, veți observa că fișierul antet *stdio.h* este inclus între paranteze ascuțite, pe când fișierul *ant_meu.h* este inclus între ghilimele. Atunci când includeți nume de fișiere antet între paranteze ascuțite, compilatorul de C va căuta respectivul fișier mai întâi în propriul său director de fișiere antet. Când compilatorul localizează fișierul, preprocesorul îl va utiliza pe acesta. Dacă nu-l întâlnește, el îl va căuta în directorul curent sau într-un director pe care îl specificați dumneavoastră. Pe de altă parte, atunci când includeți un nume de fișier între ghilimele, compilatorul va căuta numai în interiorul directorului curent.

TESTAREA DEFINIRII UNUI SIMBOL

C/C++ 149

Câteva secțiuni din acest capitol v-au prezentat simboluri predefinite de compilator. În plus, câteva secțiuni v-au arătat cum să vă definiți propriile dumneavoastră constante și macroinstrucțiuni. În funcție de programele dumneavoastră, uneori veți dori ca preprocesorul să

testeze dacă programul a definit înainte un simbol și, dacă este așa, să execute un anumit set de instrucțiuni. Pentru a testa dacă un anumit simbol a fost definit anterior, preprocesorul de C utilizează directiva *#ifdef*. Formatul directivei *#ifdef* este:

```
#ifdef simbol
  // Instructiuni
#endif
```

Atunci când preprocesorul întâlnește directiva *#ifdef*, el va testa dacă simbolul respectiv a fost definit anterior în program. Dacă da, preprocesorul va efectua instrucțiunile care urmează directivei până la instrucțiunea *#endif*. Alteori, veți dori ca preprocesorul să efectueze anumite instrucțiuni dacă programul nu a definit un anumit simbol. În aceste cazuri puteți utiliza directiva *#ifndef*. Următoarele instrucțiuni utilizează *#ifndef* pentru a indica preprocesorului să definească macroinstrucțiunea *_toupper* dacă nu a fost definită una similară:

```
#ifndef _toupper
  #define _toupper(c) ((((c) >= 'a')&&((c) <= 'z')) ? (c) - 'a'
    + 'A' : c)
#endif
```

150 PREPROCESAREA INSTRUCȚIUNILOR IF-ELSE

În secțiunea 149, ați învățat cum să utilizați instrucțiunile *#ifdef*, *#ifndef* și *#endif* pentru a cere preprocesorului să execute un set de instrucțiuni dacă un program a definit anterior (*#ifdef*) sau nu (*#ifndef*) un anumit simbol. Uneori, veți dori o procesare mai complexă, în care preprocesorul să execute un set de instrucțiuni atunci când condiția testată cu *#ifdef* este adevărată și alt set de instrucțiuni când condiția este falsă. Pentru aceasta, puteți să utilizați instrucțiunea *#else*, așa cum se arată mai jos:

```
#ifdef simbol
  // Instructiuni
#else
  // Alte instructiuni
#endif
```

De exemplu, compilatoarele Microsoft *Visual C++* și Borland *C++ 5.02* includ constante de preprocesor unice, care indică ce compilator și ce versiune utilizați pentru a compila programul. Cu ajutorul acestor constante, puteți executa secvențe specifice fiecărui compilator. De exemplu, următorul fragment de cod va afișa *Microsoft* dacă se utilizează compilatorul *Visual C++* sau *Borland* dacă se utilizează compilatorul Borland *C++ 5.02*:

```
#ifdef _MSC_VER
  printf("Microsoft");
#endif
#ifdef __BORLANDC__
  printf("Borland");
#endif
```

TESTAREA UNOR CONDIȚII DE PREPROCESOR MAI PUTERNICE

C/C++ 151

În secțiunea 149, ați învățat cum să utilizați instrucțiunile *#ifdef* și *#ifndef*, pentru a cere preprocesorului să testeze dacă un program a definit sau nu un simbol și apoi să execute instrucțiunile care urmează în funcție de rezultatul testului. În anumite cazuri, s-ar putea să doriți ca preprocesorul să testeze dacă mai multe simboluri sunt definite sau nu. Următoarele directive testează mai întâi dacă simbolul *BIBL_MEA* este definit. Dacă programul dumneavoastră a definit anterior *BIBL_MEA*, directivele de preprocesor testează dacă a fost definit și simbolul *FCT_MELE*. Dacă programul dumneavoastră nu a definit încă *FCT_MELE*, codul indică preprocesorului să includă fișierul antet *cod_meu.h*:

```
#ifdef BIBL_MEA
#ifndef FCT_MELE
#include "cod_meu.h"
#endif
#endif
```

Cu toate că directivele efectuează prelucrarea dorită, condițiile imbricate fac ca altui programator să-i fie dificil să vă urmărească intențiile. O soluție alternativă ar fi ca programul dumneavoastră să testeze definirea simbolului cu directiva *#if* și operatorul *defined*, după cum se vede mai jos:

```
#if defined(simbol)
  // Instructiuni
#endif
```

Spre deosebire de *#ifdef* și *#ifndef*, această directivă oferă avantajul posibilității de a combina testările. Următoarea directivă efectuează testul din primul exemplu al acestui capitol:

```
#if defined(BIBL_MEA) && !defined(FCT_MELE)
#include "cod_meu.h"
#endif
```

Puteți folosi *#if defined* pentru a construi condiții care utilizează operatori logici ai limbajului C (inclusiv &&, || și !).

PREPROCESAREA INSTRUCȚIUNILOR IF-ELSE ȘI ELSE-IF

C/C++ 152

În secțiunea 151, ați învățat cum se utilizează directiva către preprocesor *#if* pentru a testa dacă programul dumneavoastră a definit sau nu anterior un simbol. Când veți utiliza directiva *#if*, este posibil ca preprocesorul să execute un set de instrucțiuni când simbolul este deja definit și alt set dacă simbolul este nedefinit (*preprocesare condițională*). Puteți efectua preprocesarea condițională utilizând directiva *#else*:

```
#if defined(simbol)
  // Instructiuni
#else
```

```
    // Instructiuni
#endif
```

Ducând exemplul precedent de preprocesare cu un pas mai departe, probabil că uneori veți dori ca preprocesorul să testeze statutul altor simboluri atunci când o anumită condiție eșuează. Următoarele directive, de exemplu, indică preprocesorului să execute un set de instrucțiuni dacă simbolul *BIBL_MEA* este definit, alt set dacă acest simbol nu este definit, dar simbolul *FCT_MELE* este definit și un al treilea set dacă nici unul dintre aceste simboluri nu este definit:

```
#if defined(BIBL_MEA)
   // Instructiuni
#else if defined(FCT_MELE)
   // Instructiuni
#else
   // Instructiuni
#endif
```

Așa cum puteți vedea, directivele *#if* și *#else* vă oferă un control mai mare asupra preprocesorului.

Observație: *Unele compilatoare, inclusiv **Turbo C++ Lite**, acceptă directiva către preprocesor **#elif**, care îndeplinește aceeași funcție ca și construcția **#else if**.*

153 DEFINIREA MACROINSTRUCȚIUNILOR ȘI A CONSTANTELOR PE MAI MULTE LINII

C/C++

Câteva dintre secțiunile prezentate în acest capitol au definit constante și macroinstrucțiuni. Pe măsură ce constantele și macroinstrucțiunile dumneavoastră devin mai complexe, s-ar putea ca ele să nu încapă pe o singură linie. Atunci când definiția unei constante sau a unei macroinstrucțiuni trebuie să continue pe linia următoare, plasați un caracter *backslash* (\) la capătul liniei, ca mai jos:

```
#define sir_de_caractere_foarte_lung "Aceasta constanta de tip\
sir este foarte lunga si necesita doua linii"
#define _toupper(c) ((((c) >= 'a' ) && ((c) <= 'z' )) ? (c) -\
'a' + 'A' : c)
```

154 CREAREA PROPRIILOR MACROINSTRUCȚIUNI

C/C++

După cum ați învățat, macroinstrucțiunile vă oferă posibilitatea de a defini constante pe care preprocesorul le substituie în tot programul înainte să înceapă compilarea. În plus, macroinstrucțiunile vă permit crearea unor operații asemănătoare funcțiilor, care lucrează cu *parametri*. Parametrii sunt valori pe care le transmiteți macroinstrucțiunii. De exemplu, următoarea macroinstrucțiune, *SUMA*, returnează suma celor două valori pe care i le indicați:

```
#define SUMA(x, y) ((x) + (y))
```

Următorul program, *suma.c*, utilizează macroinstrucțiunea *SUMA* pentru a aduna mai multe valori:

```
#include <stdio.h>
#define SUMA(x, y) ((x) + (y))
```

```
void main(void)
 {
   printf("3 + 5 = %d\n", SUMA(3, 5));
   printf("3.4 + 3.1 = %f\n", SUMA(3.4, 3.1));
   printf("-100 + 1000 = %d\n", SUMA(-100, 1000));
 }
```

În cadrul definiției macroinstrucțiunii *SUMA*, *x* și *y* reprezintă parametrii macroinstrucțiunii. Atunci când transmiteți două valori macroinstrucțiunii, cum ar fi *SUMA(3, 5)*, preprocesorul substituie parametrii în cadrul macroinstrucțiunii, ca în figura 154:

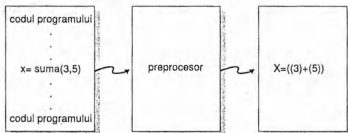


Figura 154 *Substituirea parametrilor macroinstrucțiunii* ***SUMA.***

În programul *suma.c*, substituirile vor avea ca rezultat următorul cod:

```
printf("3 + 5 = %d\n", ((3) + (5)));
printf("3.4 + 3.1 = %f\n", ((3.4) + (3.1)));
printf("-100 + 1000 = %d\n", ((-100) + (1000)));
```

UTILIZAREA CARACTERULUI PUNCT ȘI VIRGULĂ ÎN MACROINSTRUCȚIUNI

C/C++ 155

Examinând definiția macroinstrucțiunii *SUMA*, veți observa că ea nu conține caracterul punct și virgulă:

```
#define SUMA(x, y) ((x) + (y))
```

Dacă includeți punctul și virgula în cadrul macroinstrucțiunilor, preprocesorul va plasa acest caracter la fiecare apariție în cadrul programului. Să presupunem, de exemplu, că ați plasat punctul și virgula la sfârșitul definiției macroinstrucțiunii *SUMA*, așa cum se vede mai jos:

```
#define SUMA(x, y) ((x) + (y));
```

Atunci când preprocesorul expandează macroinstrucțiunea, el va include și punctul și virgula, ca mai jos:

```
printf("3 + 5 = %d\n", ((3) + (5)););
printf("3.4 + 3.1 = %f\n", ((3.4) + (3.1)););
printf("-100 + 1000 = %d\n", ((-100) + (1000)););
```

Deoarece caracterul punct și virgulă apare acum în interiorul instrucțiunii *printf* (indicând sfârșitul instrucțiunii), compilatorul va genera erori.

Observație: *În afara cazului în care doriți ca preprocesorul să includă punctul și virgula la expandare, nu utilizați punctul și virgula într-o definiție de macroinstrucțiune.*

156 CREAREA MACROINSTRUCȚIUNILOR MIN ȘI MAX

C/C++

În secțiunea 154, ați creat macroinstrucțiunea *SUMA*, care aduna două valori. Următoarele macroinstrucțiuni, *MIN* și *MAX*, returnează minimul și maximul a două valori:

```
#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))
```

Următorul program, *min_max.c*, ilustrează modul de utilizare a macroinstrucțiunilor *MIN* și *MAX*:

```
#include <stdio.h>

#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))

void main(void)
  {
    printf("Maximul valorilor 10.0 si 25.0 este %f\n",
      MAX(10.0, 25.0));
    printf("Minimul valorilor 3.4 si 3.1 este %f\n", MIN(3.4, 3.1));
  }
```

Atunci când executați programul *min_max.c*, substituirile preprocesorului vor avea ca rezultat următorul cod:

```
printf("Maximul valorilor 10.0 si 25.0 este %d\n", (((10.0)
  < (25.0)) ? (10.0): (25.0)));
printf("Minimul valorilor 3.4 si 3.1 este %f\n", (((3.4)
  > (3.1)) ? (3.4): (3.1)));
```

157 CREAREA MACROINSTRUCȚIUNILOR PATRAT ȘI CUB

C/C++

După cum ați învățat, limbajul C vă permite să definiți macroinstrucțiuni și să le transmiteți valori. Ultimele macroinstrucțiuni pe care le veți examina în această secțiune sunt *PATRAT* și *CUB*, care returnează, respectiv, pătratul (x * x) și cubul unei valori (x * x * x):

```
#define PATRAT(x) ((x) * (x))
#define CUB(x) ((x) * (x) * (x))
```

Următorul program, *pat_cub.c*, ilustrează utilizarea macroinstrucțiunilor *PATRAT* și *CUB*:

```
#include <stdio.h>
#define PATRAT(x) ((x) * (x))
```

```
#define CUB(x) ((x) * (x) * (x))
void main(void)
  {
    printf("Patratul lui 2 este %d\n", PATRAT(2));
    printf("Cubul lui 100 este %f\n", CUB(100.0));
  }
```

În acest program, substituirea va avea ca rezultat următorul cod:

```
printf("Patratul lui 2 este %d\n", ((2) * (2)));
printf("Cubul lui 100 este %f\n", ((100.0) * (100.0) * (100.0)));
```

Observație: *Pentru a preveni situația de depășire, programul **pat_cub.c** utilizează valoarea în virgulă mobilă 100.0 în cadrul macroinstrucțiunii **CUB**.*

SPAȚIILE DIN DEFINIȚIILE MACROINSTRUCȚIUNILOR

C/C++ 158

Câteva dintre secțiunile precedente v-au prezentat macroinstrucțiuni care acceptă parametri. Atunci când creați macroinstrucțiuni care acceptă parametri, trebuie să fiți atent la spațiile albe din cadrul definițiilor. Nu plasați un spațiu între numele macroinstrucțiunii și parametrii săi. De exemplu, să considerăm următoarea definiție a macroinstrucțiunii *PATRAT*:

```
#define PATRAT (x) ((x) * (x))
```

Atunci când preprocesorul vă examinează programul, spațiul de după numele macroinstrucțiunii face ca preprocesorul să presupună că trebuie să înlocuiască fiecare apariție a numelui *PATRAT* cu *(x) ((x) * (x))* în loc de *((x) * (x))*. Ca rezultat, respectiva macroinstrucțiune nu se va evalua corect și, în cele mai multe cazuri, compilatorul va genera mesaje de eroare sintactică sau mesaje de avertizare. Pentru a înțelege cum realizează preprocesorul substituirea macroinstrucțiunii, modificați programul *pat_cub.c* (prezentat în secțiunea 157) punând un spațiu după fiecare nume de macroinstrucțiune.

UTILIZAREA PARANTEZELOR

C/C++ 159

Câteva dintre secțiunile precedente v-au prezentat macroinstrucțiuni către care se transmit valori (parametri). Dacă vă uitați cu atenție la fiecare macroinstrucțiune, veți vedea că valorile sunt incluse între paranteze:

```
#define SUMA(x, y) ((x) + (y))
#define PATRAT(x) ((x) * (x))
#define CUB(x) ((x) * (x) * (x))
#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))
```

Definițiile macroinstrucțiunilor includ parametrii între paranteze pentru a accepta expresii. Ca exemplu, să considerăm următoarea instrucțiune:

```
rezultat = PATRAT(3 + 5);
```

Ar trebui ca instrucțiunea să stocheze în variabila *rezultat* valoarea 64 (8 * 8). Să presupunem, de exemplu, că ați definit macroinstrucțiunea *PATRAT* ca mai jos:

```
#define PATRAT(x) (x * x)
```

Atunci când preprocesorul substituie *x*-ul cu expresia *3* + *5*, se obține:

```
rezultat = (3 + 5 * 3 + 5);
```

Dacă ne reamintim precedența operatorilor în C, observăm că înmulțirea are precedență mai mare decât adunarea. Prin urmare, programul va calcula expresia astfel:

```
rezultat = (3 + 5 * 3 + 5);
         = (3 + 15 + 5);
         = 23;
```

Atunci când includeți fiecare parametru într-o paranteză, vă asigurați că preprocesorul va evalua corect expresia:

```
rezultat = PATRAT(3 + 5);
         = ((3 + 5) * (3 + 5));
         = ((8) * (8));
         = (64);
         = 64;
```

Observație: *Ca regulă, puneți întotdeauna parametrii macroinstrucțiunilor între paranteze.*

160 MACROINSTRUCȚIUNILE NU AU TIP



În capitolul „Funcții", veți învăța cum să creați funcții care execută anumite operații. Veți învăța că limbajul C vă permite să transmiteți valori și funcțiilor dumneavoastră, la fel cum ați transmis macroinstrucțiunilor. Dacă funcția dumneavoastră efectuează o operație și returnează un rezultat, trebuie să specificați tipul rezultatului (cum ar fi *int*, *float* etc.). De exemplu, următoarea funcție, *ad_val.c*, adună două valori întregi și returnează un rezultat de tipul *int*:

```
int ad_val(int x, int y)
  {
    return(x + y);
  }
```

În cadrul programului, puteți utiliza această funcție numai pentru a aduna două valori de tipul *int*. Dacă încercați să adunați două valori reale, va fi generată o eroare. După cum ați văzut, macroinstrucțiunile vă permit să lucrați cu valori de orice tip. Macroinstrucțiunea *SUMA*, pe care ați creat-o anterior, acceptă valori de ambele tipuri: *int* și *float*:

```
printf("3 + 5 = %d\n", SUMA(3, 5));
printf("3.4 + 3.1 = %f\n", SUMA(3.4, 3.1));
```

Atunci când utilizați macroinstrucțiuni pentru o operație aritmetică simplă, nu mai este necesară duplicarea, ca în cazul funcțiilor, pentru a permite utilizarea unor valori de tipuri

diferite. Totuși, așa cum veți învăța în capitolul „Funcții", trebuie să țineți cont și de alte aspecte atunci când decideți dacă veți utiliza macroinstrucțiuni sau funcții.

VIZUALIZAREA UNUI ȘIR

C/C++ 161

Calculatorul dumneavoastră necesită un octet de memorie pentru stocarea unui singur caracter ASCII. Așa cum ați învățat, un *șir* este o secvență de caractere ASCII. Atunci când declarați o constantă de tip șir, i se atribuie automat caracterul *NULL*. Atunci când programele își creează propriile șiruri citind caractere de la tastatură, trebuie să plaseze caracterul *NULL* la sfârșitul șirului pentru a-i marca sfârșitul. Prin urmare, cea mai bună modalitate de a vizualiza un șir de caractere este să vi-l imaginați ca pe o colecție de octeți terminată cu un caracter *NULL*, așa cum se arată în figura 161:

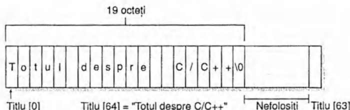


Figura 161 *Șirurile se păstrează în locații consecutive de octeți de memorie.*

Atunci când o funcție lucrează cu un șir, de obicei cunoaște numai locația la care începe acesta. Cunoscând locația de start a șirului, funcția parcurge următoarele locații de memorie până când va întâlni caracterul *NULL* (care indică sfârșitul șirului).

REPREZENTAREA UNUI ȘIR DE CARACTERE

C/C++ 162

Câteva dintre secțiunile acestei cărți utilizează constante de tip șir de caractere incluse între ghilimele, ca în exemplul următor:

```
"Totul despre C/C++"
```

Atunci când utilizați o constantă de acest tip într-un program, compilatorul de C adaugă automat caracterul *NULL* (\0) la sfârșitul șirului. Compilatorul de C va stoca această constantă în memorie așa cum se vede în figura 162.

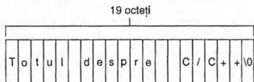


Figura 162 *Compilatorul adaugă automat caracterul* ***NULL*** *constantelor de tip șir de caractere.*

163 STOCAREA UNUI ȘIR DE CARACTERE ÎN LIMBAJUL C



Multe dintre secțiunile acestei cărți utilizează pe scară largă șirurile de caractere. De exemplu, unele programe utilizează șiruri de caractere pentru a citi fișiere sau date de la tastatură, precum și pentru alte operații. În C, un șir de caractere este un tablou de caractere terminat cu *NULL*. Pentru a crea un șir de caractere, pur și simplu veți declara un tablou de caractere, ca mai jos:

```
char sir[256];
```

Compilatorul de C va crea un șir capabil să păstreze 256 de caractere, pe care le indexează începând de la *sir[0]* până la *sir[255]*. Deoarece șirul poate conține mai puțin de 256 de caractere, compilatorul de C utilizează caracterul *NULL* (codul ASCII 0) pentru a reprezenta ultimul caracter al șirului. De obicei, limbajul C nu plasează acest caracter după ultimul caracter al șirului. În schimb, funcții cum ar fi *fgets* sau *gets* plasează acest caracter la capătul șirului de caractere. Dacă programele dumneavoastră manipulează șiruri de caractere, este responsabilitatea dumneavoastră să asigurați prezența caracterului *NULL*. Următorul program, *creezabc.c*, definește un șir de caractere de dimensiunea 256 și apoi atribuie primelor 26 de locații literele mari ale alfabetului:

```
#include <stdio.h>
void main(void)
  {
    char sir[256];
    int i;
    for (i = 0; i < 26; i++)
      sir[i] = 'A' + i;
    sir[i] = NULL;
    printf("Sirul de caractere contine %s\n", sir);
  }
```

Acest program utilizează bucla *for* pentru a atribui literele de la A la Z șirului de caractere. Apoi programul plasează caracterul *NULL* după litera Z pentru a stabili sfârșitul șirului. Funcția *printf* va afișa apoi fiecare caracter al șirului, până la caracterul *NULL*. Funcțiile limbajului C care lucrează cu șiruri de caractere utilizează caracterul *NULL* pentru a marca sfârșitul șirului. Următorul program, *a_la_j.c*, atribuie de asemenea literele de la A la Z unui șir de caractere. Totuși, programul atribuie caracterul *NULL* lui *sir[10]*, care este locația imediat următoare literei J. Atunci când *printf* afișează conținutul șirului, se va opri la litera J:

```
#include <stdio.h>
void main(void)
  {
    char sir[256];
    int i;

    for (i = 0; i < 26; i++)
      sir[i] = 'A' + i;
    sir[10] = NULL;
    printf("Sirul de caractere contine %s\n", sir);
  }
```

***Observație:** Atunci când lucrați cu șiruri de caractere, trebuie să vă asigurați că ați inclus corect caracterul **NULL** pentru a reprezenta sfârșitul șirului.*

DIFERENȚA ÎNTRE 'A' ȘI "A"

C/C++ 164

Așa cum ați învățat în secțiunea 161, un șir de caractere este o secvență de zero sau mai multe caractere ASCII pe care limbajul C o termină cu caracterul *NULL* (ASCII 0). Atunci când lucrați cu caractere în C, puteți utiliza valoarea numerică în ASCII a caracterului sau puteți plasa caracterul între apostrofuri, cum ar fi 'A'. Pe de altă parte, când utilizați ghilimele, cum ar fi "A", compilatorul de C creează un șir de caractere care conține litera specificată (sau literele) și termină șirul cu caracterul *NULL*. Figura 164 ilustrează stocarea în C a constantelor 'A' și "A".

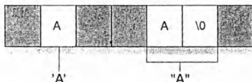


***Figura 164** Stocarea în C a constantelor 'A' și "A".*

Deoarece sunt stocate în mod diferit, constantele de tip caracter și cele de tip șir de caractere nu sunt similare și trebuie să aveți grijă să le tratați în mod diferit în programele dumneavoastră.

REPREZENTAREA UNEI GHILIMELE ÎNTR-O CONSTANTĂ DE TIP ȘIR DE CARACTERE

C/C++ 165

Așa cum ați învățat, pentru a crea o constantă de tip șir de caractere, în program trebuie plasate caracterele dorite între ghilimele:

```
"Aceasta este o constanta de tip sir de caractere"
```

În anumite programe, poate că veți dori ca o constantă de tip șir de caractere să conțină caracterul ghilimele. De exemplu, să presupunem că trebuie să reprezentați următorul șir:

```
"Stop!", spuse ea.
```

Deoarece limbajul C utilizează ghilimelele pentru a defini constantele de tip șir de caractere, trebuie să existe o modalitate să-i spuneți compilatorului că doriți să includeți ghilimelele în cadrul șirului. Pentru aceasta, utilizați secvența escape \", cum se vede mai jos:

```
"\"Stop!\", spuse ea."
```

Următorul program, *ghil.c*, utilizează secvența escape \" pentru a plasa ghilimelele într-o constantă de tip șir de caractere:

```
#include <stdio.h>

void main(void)
   {
```

```
    char sir[] = "\"Stop!\", spuse ea.";
    printf(sir);
  }
```

166 DETERMINAREA LUNGIMII UNUI ȘIR DE CARACTERE



În secțiunea 163, ați învățat că funcțiile limbajului C utilizează în mod obișnuit caracterul *NULL* pentru a reprezenta sfârșitul unui șir de caractere. Funcțiile cum ar fi *fgets* sau *cgets* atribuie caracterul *NULL* pentru a indica sfârșitul șirului. Următorul program, *un_sir.c*, utilizează funcția *gets* pentru a citi un șir de caractere de la tastatură. Programul folosește apoi bucla *for* pentru a afișa caracterele șirului unul câte unul, până când condiția *sir[i] != NULL* este evaluată ca falsă:

```
#include <stdio.h>

void main(void)
  {
    char sir[256];   // Sir introdus de utilizator
    int i;           // Indexul sirului
    printf("Introduceti un sir de caractere si apoi apasati
      Enter:\n");
    gets(sir);
    // Afiseaza fiecare caracter pana cand gaseste NULL
    for (i = 0; sir[i] != NULL; i++)
      putchar(sir[i]);
    printf("\nNumarul de caractere in sir este %d\n", i);
  }
```

167 UTILIZAREA FUNCȚIEI STRLEN



Când veți lucra cu șiruri de caractere în cadrul programelor dumneavoastră, veți executa multe operații bazate pe numărul de caractere din șir. Pentru a vă ajuta să determinați numărul de caractere dintr-un șir, cele mai multe compilatoare de C oferă funcția *strlen*, care returnează numărul de caractere din șir. Formatul funcției *strlen* este următorul:

```
#include <string.h>
size_t strlen(const char *sir);
```

Următorul program, *strlen.c*, ilustrează utilizarea funcției *strlen*:

```
#include <stdio.h/>
#include <string.h>

void main(void)
  {
    char titlu_carte[] = "Totul despre C/C++";

   printf("%s contine %d caractere\n", titlu_carte,
```

```
    strlen(titlu_carte));
}
```

Atunci când compilați și executați programul *strlen.c*, pe ecranul dumneavoastră se va afișa:

```
Totul despre C/C++ contine 35 caractere
C:\>
```

Pentru a înțelege mai bine cum operează funcția *strlen*, să analizăm următoarea implementare. Funcția numără caracterele dintr-un șir până la caracterul *NULL* fără a-l pune și pe el la socoteală:

```
size_t strlen(const char *sir)
  {
    int i = 0;
    while (sir[i])
      i++;
    return(i);
  }
```

COPIEREA UNUI ȘIR DE CARACTERE ÎN ALTUL

C/C++ 168

Atunci când programele dumneavoastră lucrează cu șiruri de caractere, este posibil să aveți nevoie să copiați conținutul unui șir de caractere în alt șir. Pentru a vă ajuta să efectuați această operație cu șiruri, cele mai multe compilatore de C vă oferă funcția *strcpy*, care copiază caracterele dintr-un șir (parametrul *sursa*) într-un alt șir (parametrul *destinatie*):

```
#include <string.h>
char *strcpy(char *destinatie, const char *sursa);
```

Funcția *strcpy* returnează un pointer care indică începutul șirului destinație. Următorul program, *strcpy.c*, ilustrează modul în care puteți folosi funcția *strcpy.c* în cadrul programelor dumneavoastră:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char titlu[] = "Totul despre C/C++";
    char carte[128];
    strcpy(carte, titlu);
    printf("Numele cartii e %s\n", carte);
  }
```

Pentru a înțelege mai bine cum operează funcția *strcpy*, să studiem următorul exemplu:

```
char *strcpy(char *destinatie, const char *sursa)
  {
    char * initial = destinatie;

    while (*destinatie++ = *sursa++)
      ;
```

```
    return(initial)
  }
```

Funcția *strcpy* copiază pur și simplu literele din șirul sursă în șirul destinație, până la caracterul *NULL*, pe care îl include.

169 ADĂUGAREA CONȚINUTULUI UNUI ȘIR LA ALT ȘIR

C/C++

Atunci când programele dumneavoastră lucrează cu șiruri de caractere, este posibil să aveți nevoie să adăugați unui șir conținutul altuia. De exemplu, dacă un șir de caractere conține numele unui subdirector și altul un nume de fișier, puteți adăuga numele fișierului la numele subdirectorului pentru a crea în întregime un nume de cale (*path*). Programatorii de C numesc *concatenare* de șiruri procesul de adăugare a unui șir de caractere la altul. Pentru a vă ajuta să adăugați un șir de caractere la altul, cele mai multe compilatoare de C oferă funcția *strcat*, care concatenează (adaugă) un șir de caractere la un șir țintă, cum se vede mai jos:

```
#include <string.h>
char *strcat(char *destinatie, const char *sursa);
```

Următorul program, *strcat.c*, ilustrează utilizarea funcției *strcat*:

```
#include <stdio.h>
#include <string.h>
void main(void)
  {
    char nume[64] = "Totul despre";
    strcat(nume, "C/C++");
    printf("Numele cartii este %s\n", nume);
  }
```

Atunci când compilați și executați programul *strcat.c*, pe ecranul dumneavoastră se vor afișa următoarele:

```
Numele cartii este Totul despre C/C++
C:\>
```

Pentru a înțelege mai bine cum operează funcția *strcat*, analizați următoarea implementare:

```
char *strcat(char *destinatie, const char *sursa)
{
   char *initial = destinatie;

   while (*destinatie)
     destinatie++;     // Cauta finalul sirului
   while (*destinatie++ = *sursa++)
     ;
   return (initial);
}
```

După cum puteți vedea, funcția *strcat* ciclează șirul de caractere destinație până când întâlnește caracterul *NULL*, pe care îl include în șirul destinație.

ADĂUGAREA A N CARACTERE LA UN ȘIR

C/C++ 170

În secțiunea 169, ați învățat că funcția *strcat* permite adăugarea (concatenarea) unui șir de caractere la altul. În unele cazuri, doriți să adăugați la șirul destinație toate caracterele altui șir, iar alteori doar primele două, trei sau *n* caractere. Pentru a facilita adăugarea a *n* caractere la un șir, cele mai multe compilatoare de C oferă funcția *strncat*, care adaugă primele *n* caractere ale șirului sursă la șirul destinație, ca mai jos:

```
#include <string.h>
char *strncat(char *destinatie, const char *sursa, size_t n);
```

Dacă *n* precizează un număr de caractere mai mare decât numărul de caractere din șirul sursă, funcția *strncat* va copia toate caracterele până la sfârșitul șirului, nu mai mult. Programul următor, *srtncat.c*, ilustrează modul de utilizare a funcției *strncat*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char nume[64] = "Bill";
    strncat(nume, " si Hillary", 3);
    printf("Ai votat cu %s?\n", nume);
  }
```

Atunci când compilați și executați programul *strncat.c*, pe ecran va apărea următorul text:

```
Ai votat cu Bill si?
C:\>
```

Pentru a înțelege mai ușor cum operează funcția *strncat*, analizați următoarea implementare:

```
char *strncat(char *destinatie, const char *sursa, int n)
  {
    char *initial = destinatie;
    int i = 0;
    while (*destinatie)
      destinatie++;
    while ((i++ < n) && (*destinatie++ = *sursa++))
      ;
    if (i > n)
      *destinatie = NULL;
    return (initial);
  }
```

TRANSFORMAREA UNUI ȘIR DE CARACTERE ÎN ALT ȘIR

C/C++ 171

Multe secțiuni din această carte au arătat modalități de copiere a unui șir de caractere în altul. Funcția *strxfrm* copiază conținutul unui șir de caractere în alt șir până la numărul de caractere precizat în parametrul *n* și apoi returnează lungimea șirului rezultat.

```
#include <string.h>
size_t strxfrm(char *destinatie, char *sursa, size_t n);
```

Parametrul *destinatie* este un *pointer* la care funcția *strxfrm* copiază șirul sursă. Parametrul *n* precizează numărul maxim de caractere de copiat. Următorul program, *strxfrm.c*, ilustrează modul de utilizare a funcției *strxfrm*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char buffer[64] = "Totul despre C/C++";
    char destinatie[64];
    int lungime;

    lungime = strxfrm(destinatie, buffer, sizeof(buffer));
    printf("Lungimea %d Destinatie %s Buffer %s\n", lungime,
      destinatie, buffer);
  }
```

172 LIMITELE UNUI ȘIR DE CARACTERE



Multe secțiuni ale acestui capitol au prezentat funcții care copiază sau adaugă caractere de la un șir la altul. Atunci când efectuați operații cu șiruri de caractere, trebuie să vă asigurați că nu depășiți locațiile de memorie alocate șirului. Pentru a vedea care sunt problemele legate de scrierea peste limitele șirurilor, să luăm de exemplu următoarea declarație, care creează un șir capabil să păstreze 10 caractere:

```
char sir[10]
```

Dacă atribuiți șirului mai mult de 10 caractere, s-ar putea ca sistemul dumneavoastră de operare să nu sesizeze eroarea și să suprascrie caracterele suplimentare peste locațiile de memorie corespunzătoare altor variabile. Nu numai că este foarte dificil de corectat o eroare de suprascriere, dar o astfel de eroare poate, de asemenea, să cauzeze blocarea execuției atât a programului dumneavoastră, cât și a sistemului de operare. Ca regulă, declarați șirurile de caractere puțin mai mari decât vă gândiți că veți avea nevoie. Făcând aceasta, veți reduce probabilitatea suprascrierii șirurilor. Dacă programul dumneavoastră generează din când în când erori, examinați codul pentru a vedea dacă nu cumva se produce o astfel de suprascriere.

173 TESTAREA IDENTITĂȚII A DOUĂ ȘIRURI DE CARACTERE



Când veți crea programe care lucrează cu șiruri de caractere, adesea veți compara două șiruri pentru a verifica dacă sunt identice. Când vreți să stabiliți dacă două șiruri conțin aceleași caractere, puteți folosi funcția *streql*, cum se vede mai jos:

```
int streql(char *str1, char *str2)
  {
```

```
   while ((*str1 == *str2) && (*str1))
     {
       str1++;
       str2++;
     }
   return((*str1 == NULL) && (*str2 == NULL));
 }
```

Funcția *streql* returnează valoarea 1 dacă cele două șiruri sunt egale și valoarea 0 dacă ele nu sunt egale. Următorul program, *streql.c*, ilustrează felul în care se utilizează funcția *streql*:

```
#include <stdio.h>

void main(void)
  {
    printf("Testare Abc si Abc %d\n", streql("Abc", "Abc"));
    printf("Testare abc si Abc %d\n", streql("abc", "Abc"));
    printf("Testare abcd si abc %d\n", streql("abcd", "abc"));
  }
```

Atunci când compilați și executați programul *streql.c*, pe ecranul dumneavoastră se va afișa următorul rezultat:

```
Test Abc si Abc 1
Test abc si Abc 0
Test abcd si abc 0
C:\>
```

IGNORAREA DIFERENȚEI DINTRE MAJUSCULE ȘI MINUSCULE CÂND SE COMPARĂ DOUĂ ȘIRURI

C/C++174

În secțiunea 173, ați creat funcția *streql*, care permite programului dumneavoastră să determine dacă două șiruri sunt egale. Atunci când funcția *streql* compară două șiruri de caractere, ea consideră majusculele diferite de minuscule. Este posibil să doriți să comparați două șiruri fără să luați în considerare această distincție. Pentru a compara două șiruri de caractere fără a ține seama dacă literele sunt mari sau mici, puteți crea funcția *strieql*, așa cum se vede mai jos:

```
#include <ctype.h>

int strieql(char *str1, char *str2)
  {
    while ((toupper(*str1) == toupper(*str2)) && (*str1))
      {
        str1++;
        str2++;
      }
    return((*str1 == NULL) && (*str2 == NULL));
  }
```

După cum puteţi vedea, funcţia *strieql* converteşte fiecare caracter din fiecare şir de caractere în majuscule înainte de a compara cele două şiruri. Următorul program, *strieql.c*, ilustrează utilizarea funcţiei *strieql*:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
  {
    printf("Testare Abc si Abc %d\n", strieql("Abc", "Abc"));
    printf("Testare abc si Abc %d\n", strieql("abc", "Abc"));
    printf("Testare abcd si abc %d\n", strieql("abcd", "abc"));
  }
```

Atunci când compilaţi şi executaţi programul *strieql.c*, pe ecranul dumneavoastră se vor afişa următoarele:

```
Test Abc si Abc 1
Test abc si Abc 1
Test abcd si abc 0
C:\>
```

175 CONVERTIREA CARACTERELOR UNUI ŞIR ÎN MAJUSCULE SAU MINUSCULE



Atunci când programele dumneavoastră lucrează cu şiruri de caractere, veţi dori uneori să convertiţi un şir de caractere în majuscule. De exemplu, când utilizatorul introduce numele unui fişier sau al clientului, e posibil ca programul să convertească întregul şir de caractere în majuscule, pentru a simplifica operaţia de comparare a şirurilor sau pentru a fi sigur că programul păstrează datele într-un format unitar. Pentru a vă ajuta să realizaţi această conversie, cele mai multe compilatoare de C pun la dispoziţie funcţiile *strlwr* şi *strupr*, prezentate mai jos:

```
#include <string.h>

char *strlwr(char *sir);
char *strupr(char *sir);
```

Următorul program, *strcase.c*, ilustrează utilizarea funcţiilor *strlwr* şi *strupr*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    printf(strlwr("Totul despre C/C++\n"));
    printf(strupr("Totul despre C/C++\n"));
  }
```

Pentru a înţelege mai bine aceste două funcţii, să analizăm următoarea implementare a funcţiei *strlwr*:

```
#include <ctype.h>
char *strlwr(char *sir)
  {
    char *initial = sir;
    while (*sir)
      {
       *sir = tolower(*sir);
       sir++;
      }
    return(initial);
  }
```

După cum puteți vedea, ambele funcții prezentate, *strlwr* și *strupr*, ciclează șirul de caractere convertind fiecare caracter fie în majuscule, fie în minuscule, conform funcției invocate.

OBȚINEREA PRIMEI APARIȚII A UNUI CARACTER ÎN ȘIR

C/C++ 176

Când veți crea programe care lucrează cu șiruri de caractere, puteți să găsiți prima apariție din stânga a unui anumit caracter în interiorul șirului. De exemplu, dacă lucrați cu un șir care conține numele unei căi, puteți să căutați primul caracter *backslash* (\) din șir. Pentru a vă ajuta să căutați prima apariție a unui caracter într-un șir, cele mai multe compilatoare pun la dispoziție funcția numită *strchr*, care returnează un pointer la prima apariție în șir a caracterului specificat, ca mai jos:

```
#include <string.h>
char *strchr(const char *sir, int caracter);
```

Dacă funcția *strchr* nu găsește caracterul specificat în interiorul șirului, ea returnează pointerul caracterului *NULL* care marchează sfârșitul șirului. Următorul program, *strchar.c*, ilustrează utilizarea funcției *strchr*.

```
#include <stdio.h>
#include <string.h>
void main(void)
  {
    char titlu[64] = "Totul despre C/C++";
    char *ptr;
    ptr = strchr(titlu, 'm');
    if (*ptr)
      printf("Prima aparitie a lui C este la deplasamentul %d\n",
        ptr - titlu);
    else
      printf("Caracterul nu exista\n");
  }
```

Atunci când compilaţi şi executaţi programul *strchr.c*, pe ecranul dumneavoastră va fi afişat următorul rezultat:

```
Prima aparitie a lui C este la deplasamentul 8
C:\>
```

Trebuie să observaţi că *strchr* nu conţine indexul primei apariţii a unui caracter, ci un pointer. Pentru a înţelege mai bine funcţia *strchr*, să analizăm următoarea implementare:

```
char *strchr(const char *sir, int litera)
  {
    while ((*sir != litera) && (*sir))
      sir++;
    return(sir);
  }
```

177 RETURNAREA INDEXULUI PRIMEI APARIŢII DINTR-UN ŞIR



În secţiunea 176, aţi învăţat modul în care se foloseşte funcţia *strchr* pentru a obţine un pointer la prima apariţie a unui caracter în interiorul şirului. Dacă însă trataţi şirul de caractere ca tablou, probabil că veţi prefera să lucraţi cu un index al caracterelor, nu cu un pointer. Puteţi să folosiţi funcţia *strchr* pentru a obţine indexul caracterului dorit, scăzând adresa de început a şirului din pointerul pe care îl returnează funcţia *strchr*, aşa cum se vede mai jos:

```
char_ptr = strchr(sir, caracter);
index = char_ptr - sir;
```

Dacă funcţia *strchr* nu găseşte caracterul în şir, atunci valoarea pe care ea o atribuie indexului va fi egală cu lungimea şirului. O altă metodă este utilizarea funcţiei *str_index*, cum se vede mai jos:

```
int str_index(const char *sir, int *litera)
  {
    char *initial = sir;

    while ((*sir != litera) && (*sir))
      sir++;
    return(sir - initial);
  }
```

178 GĂSIREA ULTIMEI APARIŢII A UNUI CARACTER ÎNTR-UN ŞIR



Când veţi crea programe care lucrează cu şiruri de caractere, puteţi să găsiţi ultima apariţie (cea din extremitatea dreaptă) a unui anumit caracter al şirului. De exemplu, dacă lucraţi cu un şir care conţine un nume de cale, puteţi să căutaţi ultimul caracter *backslash* (\) din şir, cu scopul de a găsi locaţia unde începe numele fişierului. Pentru a vă ajuta să căutaţi ultima

apariție a unui caracter într-un șir, cele mai multe compilatoare pun la dispoziție funcția numită *strrchr*, care returnează un pointer la ultima apariție în șir a caracterului specificat, ca mai jos:

```
#include <string.h>

char *strrchr(const char *sir, int caracter);
```

Dacă funcția *strrchr* nu găsește în interiorul șirului caracterul specificat, ea returnează pointerul la caracterul *NULL*, care marchează sfârșitul șirului. Următorul program, *strrchr.c*, ilustrează modul de utilizare a funcției *strrchr*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char titlu[64] = "Totul despre C/C++";
    char *ptr;

    if (ptr = strrchr(titlu, 'C'))
      printf("Ultima aparitie a lui C e la deplasamentul %d\n",
      ptr - titlu);
    else
      printf("Caracterul nu exista \n");
  }
```

Trebuie să observați că *strrchr* nu conține indexul ultimei apariții a unui caracter, ci un pointer la respectivul caracter. Pentru a înțelege mai bine funcția *strrchr*, să analizăm următoarea implementare:

```
char *strrchr(const char *sir, int litera)
  {
    char *ptr = NULL

    while (*sir)
      {
        if (*sir == litera)
          ptr = sir;
        sir++;
      }
    return(ptr);
  }
```

RETURNAREA INDEXULUI ULTIMEI APARIȚII DINTR-UN ȘIR

C/C++ 179

În secțiunea 178, ați învățat cum se folosește funcția *strrchr* pentru a obține un pointer la ultima apariție a unui caracter într-un șir. Dacă tratați șirul de caractere ca un tablou, probabil că veți prefera să lucrați cu un index al caracterelor, nu cu un pointer. Puteți să folosiți

funcţia *strrchr* pentru a obţine indexul caracterului dorit, scăzând adresa de început a şirului din pointerul pe care îl returnează funcţia *strrchr*:

```
char_ptr = strrchr(sir, caracter);
index = char_ptr - sir;
```

Dacă funcţia *strrchr* nu găseşte caracterul în interiorul şirului, valoarea pe care o va atribui indexului va fi egală cu lungimea şirului de caractere. O altă metodă este utilizarea funcţiei *strr_index*, cum se vede mai jos:

```
int strr_index(const char *sir, int litera)
  {
    char *initial = sir;
    char *ptr = NULL;
    while (*sir)
      {
        if (*sir == litera)
          ptr = sir;
        sir++;
      }
    return((*ptr) ? ptr - initial : sir - initial);
  }
```

180 ŞIRURILE FAR



Aşa cum se poate vedea în capitolul „Memoria", pointerii *far* permit programelor scrise pentru sistemul de operare DOS să acceseze date situate în afara segmentului curent de date, de 64 KB. Atunci când lucraţi cu pointeri *far*, trebuie să folosiţi, de asemenea, funcţii care aşteaptă ca parametrii lor să fie pointeri *far*. Din păcate, nici una dintre rutinele de manipulare a şirurilor de caractere prezentate în această secţiune nu aşteaptă pointeri *far* către şiruri. Transmiterea unui pointer *far* către una dintre funcţiile de manipulare a şirurilor de caractere prezentate în această secţiune va provoca apariţia unei erori. Totuşi, multe compilatoare dispun de implementări ale acestor funcţii destinate pointerilor *far*. De exemplu, pentru a determina lungimea unui şir de caractere la care face referire un pointer *far*, puteţi utiliza funcţia *_fstrlen*, prezentată mai jos:

```
#include <string.h>
size_t _fstrlen(const char *sir)
```

Pentru a determina care funcţii *far* sunt acceptate de compilatorul dumneavoastră, studiaţi documentaţia aferentă acestuia.

Observaţie: *Aşa cum aţi învăţat anterior,* ***Visual C++*** *nu acceptă declaraţii* ***far*** *(nici pointeri, nici funcţii), astfel că, în* ***Visual C++****, puteţi utiliza funcţia* ***strlen*** *cu pointeri* ***char*** *de orice dimensiune.*

SCRIEREA FUNCȚIILOR PENTRU ȘIRURI FAR

C/C++ 181

În secțiunea 180, ați învățat că unele compilatoare pun la dispoziție funcții care acceptă șiruri de caractere la care face referire de pointeri *far*. Dacă utilizați un compilator care nu dispune de astfel de funcții, puteți crea singuri funcții pentru șiruri *far* modificând funcțiile prezentate în această secțiune. De exemplu, următoarea funcție, *fstreql*, ilustrează o implementare bazată pe pointeri *far* a funcției *streql* (în locul celei standard, bazate pe pointeri locali):

```
int fstreql(char far *sirl, char far *sir2)
  {
    while ((*sirl == *sir2) && (*sirl))
     {
       sirl++;
       sir2++;
     }
    return((*sirl == NULL) && (*sir2 == NULL));
  }
```

Observație: *Așa cum ați învățat anterior,* ***Visual C++*** *nu suportă declarații* ***far****, astfel că, în* ***Visual C++****, puteți utiliza funcția* ***streql*** *cu pointeri* ***char*** *de orice dimensiune.*

NUMĂRAREA APARIȚIILOR UNUI CARACTER ÎNTR-UN ȘIR

C/C++ 182

Când creați programe care lucrează cu șiruri de caractere, pot apărea situații în care doriți să știți de câte ori apare un caracter într-un șir. Pentru a număra aparițiile unui caracter într-un șir, puteți folosi funcția *charcnt*, ca mai jos:

```
int charcnt(const char *sir, int litera)
  {
    int nr = 0;
    while (*sir)
     if (*sir == litera)
       nr++;
    return(nr);
  }
```

INVERSAREA CONȚINUTULUI UNUI ȘIR

C/C++ 183

Când creați programe care lucrează cu șiruri de caractere, pot apărea situații în care doriți să inversați ordinea caracterelor dintr-un șir. Pentru a simplifica această operație, cele mai multe compilatoare oferă posibilitatea utilizării funcției *strrev*, ca mai jos:

```
#include <string.h>
char *strrev(char *sir);
```

Pentru a înțelege mai bine funcția *strrev*, analizați următoarea implementare:

```
char *strrev(char *sir)
  {
    char *initial = sir;
    char *urmator = sir;
    char temp;
    while (*sir)
      sir++;
    while (urmator < sir)
      {
        temp = *(--sir);
        *sir = *urmator;
        *urmator++ = temp;
      }
    return(initial);
  }
```

184 ATRIBUIREA UNUI CARACTER SPECIFICAT UNUI ȘIR ÎNTREG DE CARACTERE



Când creați programe care lucrează cu șiruri de caractere, pot apărea situații în care doriți să înlocuiți toate carcterele unui șir cu un anumit caracter. De exemplu, uneori vreți să suprascrieți valoarea curentă a unui șir înainte de a-l transmite către o funcție. Pentru a simplifica suprascrierea fiecărui caracter din șir, cele mai multe compilatoare de C oferă funcția *strset*, care înlocuiește fiecare caracter din șir cu un caracter specificat, ca mai jos:

```
#include <string.h>

char *strset(char *sir, int litera);
```

Funcția *strset* atribuie caracterul specificat fiecărei locații a șirului până când întâlnește caracterul *NULL*. Pentru a înțelege mai bine funcția *strset*, analizați următoarea implementare:

```
char *strset(char *sir, int litera)
  {
    char *initial = sir;

    while (*sir)
      *sir++ = litera;
    return(initial);
  }
```

După cum vedeți, funcția parcurge șirul de caractere și atribuie caracterul specificat până când întâlnește caracterul *NULL*.

185 COMPARAREA A DOUĂ ȘIRURI DE CARACTERE



În secțiunea 173, ați creat funcția *streql*, care permite programelor dumneavoastră să testeze dacă două șiruri de caractere sunt sau nu egale. În funcție de activitatea pe care trebuie să o

efectueze programul dumneavoastră, pot apărea situații în care trebuie să știți dacă un șir este mai mare decât altul (de exemplu, când programul execută o operație de sortare). Pentru a ajuta programele dumneavoastră să execute această operație care determină valoarea diferitelor șiruri, cele mai multe compilatoare de C furnizează o funcție numită *strcmp*, care compară două șiruri de caractere, ca mai jos:

```
#include <string.h>

int strcmp(const char *sir1, const char *sir2);
```

Dacă șirurile sunt egale, funcția *strcmp* returnează valoarea 0. Dacă primul este mai mare decât al doilea, returnează o valoare mai mică decât 0. Dacă al doilea șir este mai mare decât primul, funcția *strcmp* returnează o valoare mai mare decât 0. Următorul program, *strcmp.c*, ilustrează modul în care se utilizează funcția *strcmp*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    printf("Compara Abc cu Abc %d\n", strcmp("Abc", "Abc"));
    printf("Compara abc cu Abc %d\n", strcmp("abc", "Abc"));
    printf("Compara abcd cu abc %d\n", strcmp("abcd", "abc"));
    printf("Compara Abc cu Abcd %d\n", strcmp("Abc", "Abcd"));
    printf("Compara abcd cu abce %d\n", strcmp("abcd", "abce"));
    printf("Compara Abce cu Abcd %d\n", strcmp("Abce", "Abcd"));
  }
```

Pentru a înțelege mai bine funcția *strcmp*, să analizăm implementarea următoare:

```
int strcmp(const char *s1, const char *s2);
  {
    while ((*s1 == *s2) && (*s1))
      {
        s1++;
        s2++;
      }
    if ((*s1 == *s2) && (! *s1)) // Aceleasi siruri
      return(0);
    else if ((*s1) && (! *s2)) // Aceleasi dar s1 mai mare
      return(-1);
    else if ((*s2) && (! *s1)) // Aceleasi dar s2 mai mare
      return(1);
    else
      return((*s1 > *s2) ? -1: 1); // Diferite
  }
```

186 COMPARAREA PRIMELOR N CARACTERE DIN DOUĂ ŞIRURI

C/C++

În secţiunea 185, aţi învăţat cum se foloseşte funcţia *strcmp* pentru a compara două şiruri de caractere. În funcţie de programele dumneavoastră, pot apărea situaţii în care nu trebuie să comparaţi decât primele *n* caractere din două şiruri. Pentru a facilita compararea a *n* caractere din două şiruri, cele mai multe compilatoare de C oferă o funcţie numită *strncmp*, cum se vede mai jos:

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

La fel ca *strcmp*, funcţia *strncmp* returnează valoarea 0 dacă şirurile sunt egale şi valori mai mici sau mai mari decât 0, după cum primul şir este mai mare sau mai mic decât celălalt. Următorul program, *strncmp*, ilustrează utilizarea funcţiei *strncmp*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    printf("Compara 3 litere Abc cu Abc %d\n", strncmp("Abc",
      "Abc", 3));
    printf("Compara 3 litere abc cu Abc %d\n", strncmp("abc",
      "Abc", 3));
    printf("Compara 3 litere abcd cu abc %d\n",
      strncmp("abcd", "abc", 3));
    printf("Compara 5 litere Abc cu Abcd %d\n",
        strncmp("Abc", "Abcd", 5));
    printf("Compara 4 litere abcd cu abce %d\n",
        strncmp("abcd", "abce", 4));
  }
```

Pentru a înţelege mai bine funcţia *strncmp*, să analizăm următoarea implementare:

```
int strncmp(const char *s1, const char *s2, int n);
  {
    int i = 0;
    while ((*s1 == *s2) && (*s1) && i < n)
      {
        s1++;
        s2++;
        i++;
      }
    if (i == n)     // Aceleasi siruri
      return(0);
    else if ((*s1 == *s2) && (! *s1)) // Aceleasi siruri
      return(0);
    else if ((*s1) && (! *s2)) // Aceleasi dar s1 mai mare
```

```
        return(-1);
    else if ((*s2) && (! *s1)) // Aceleasi dar s2 mai mare
        return(1);
    else
        return((*s1 > *s2) ? -1: 1);
}
```

COMPARAREA ȘIRURILOR FĂRĂ A FACE DIFERENȚA ÎNTRE LITERELE MARI ȘI MICI

C/C++ 187

În secțiunea 185, ați învățat cum se utilizează funcția *strcmp* pentru a compara două șiruri de caractere. De asemenea, în secțiunea 185 ați învățat cum se folosește funcția *strncmp* pentru a compara primele *n* caractere din două șiruri. Ambele funcții, *strcmp* și *strncmp*, consideră literele mari și mici ca fiind diferite. În funcție de programele dumneavoastră, puteți ca la compararea unor șiruri de caractere să fie ignorată această diferență. Pentru o asemenea operație, cele mai multe compilatoare de C dispun de funcțiile *stricmp* și *strncmpi*, prezentate mai jos:

```
#include <string.h>
int stricmp(const char s1, const char s2);
int strncmpi(const char *s1, const char *s2, size_t n);
```

Următorul program, *compi.c*, ilustrează modul de utilizare a funcțiilor *stricmp* și *strncmpi*:

```
#include <stdio.h>
#include <string.h>
void main(void)
  {
    printf("Compara Abc cu Abc %d\n", stricmp("Abc", "Abc"));
    printf("Compara abc cu Abc %d\n", stricmp("abc", "Abc"));
    printf("Compara 3 litere abcd cu ABC %d\n",
        strncmpi("abcd", "ABC", 3));
    printf("Compara 5 litere abc cu Abcd %d\n",
        strncmpi("abc", "Abcd", 5));
  }
```

Atunci când compilați și executați programul *compi.c*, pe ecran va apărea:

```
Compara Abc cu Abc 0
Compara abc cu Abc 0
Compara 3 litere abcd cu ABC 0
Compara 5 litere abc cu Abcd -1
C:\>
```

188 CONVERTIREA UNEI REPREZENTĂRI DE TIP ŞIR ÎNTR-UN NUMĂR



Atunci când programele dumneavoastră lucrează cu şiruri de caractere, una dintre cele mai frecvente operaţii pe care trebuie să o executaţi este convertirea reprezentării ASCII a unei valori într-o valoare numerică. De exemplu, dacă cereţi utilizatorului să-şi introducă de la tastatură salariul, va trebui să convertiţi şirul de caractere introdus într-o valoare reală în virgulă mobilă. Pentru a vă ajuta să convertiţi valorile ASCII, cele mai multe compilatoare de C pun la dispoziţie un set de funcţii de bibliotecă run-time. Tabelul 188 descrie pe scurt funcţiile standard de conversie a reprezentărilor ASCII.

Funcţie	Utilitate
atof	Converteşte o reprezentare de tip şir de caractere a unei valori reale în virgulă mobilă
atoi	Converteşte o reprezentare de tip şir de caractere a unei valori întregi
atol	Converteşte o reprezentare de tip şir de caractere a unei valori întreagi de tip long
strtod	Converteşte o reprezentare a unui şir de caractere a unei valori reale în dublă precizie
strtol	Converteşte o reprezentare de tip şir de caractere a unei valori de tip long

Tabelul 188 *Funcţiile de bibliotecă run-time pe care programul dumneavoastră le poate folosi pentru conversia reprezentărilor ASCII în valori numerice.*

Următorul program, *asciinum.c*, ilustrează utilizarea acestor funcţii de conversie:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    int rezultat_int;
    float rezultat_float;
    long rezultat_long;

     rezultat_int = atoi("1234");
    rezultat_float = atof("12345.678");
    rezultat_long = atol("1234567L");
    printf("%d %f %ld\n", rezultat_int, rezultat_float,
      rezultat_long);
  }
```

189 DUPLICAREA CONŢINUTULUI UNUI ŞIR DE CARACTERE



Atunci când programele dumneavoastră lucrează cu şiruri de caractere, este necesar uneori să duplicaţi rapid conţinutul unui şir. Dacă în unele situaţii programul dumneavoastră va trebui să copieze şirul, iar în altele nu, e posibil ca în timpul execuţiei programul să aloce în

mod *dinamic* memoria necesară pentru a păstra copia şirului. Pentru a permite programelor să aloce (dinamic) memorie în timpul execuţiei atunci când trebuie copiat un şir de caractere, cele mai multe compilatoare de C oferă funcţia *strdup*, prezentată mai jos:

```
#include <string.h>

char *strdup(const char *un_sir);
```

Când invocaţi funcţia *strdup*, aceasta utilizează *malloc* pentru a aloca memoria şi apoi copiază şirul de caractere în locaţia de memorie. După ce programul a terminat de folosit copia şirului, el poate elibera memoria utilizând instrucţiunea *free*. Programul care urmează, *strdup.c*, ilustrează utilizarea funcţiei *strdup*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char *titlu;

    if ((titlu = strdup("Totul despre C/C++")))
      printf("Titlu: %s\n", titlu);
    else
      printf("Eroare la duplicarea sirului");
  }
```

Pentru a înţelege mai bine funcţia *strdup*, să urmărim următoarea implementare:

```
#include <string.h>
#include <malloc.h>

char *strdup(const char *s1)
  {
    char *ptr;

    if ((ptr = malloc(strlen(s1) + 1))) // Aloca buffer
      strcpy(ptr, s1);
    return(ptr);
  }
```

GĂSIREA PRIMEI APARIŢII A UNUI CARACTER

C/C++190

În secţiunea 176, aţi învăţat cum să utilizaţi funcţia *strchr* pentru a găsi prima apariţie a unui anumit caracter. În funcţie de programul dumneavoastră, puteţi să căutaţi într-un şir de caractere prima apariţie a oricărui caracter dintr-un anumit set. Pentru a vă ajuta să căutaţi într-un şir orice caracter dintr-o mulţime dată, cele mai multe compilatoare de C oferă funcţia *strspn*, prezentată mai jos:

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

Funcția returnează indicele primului caracter din șirul s1 care nu e conținut în șirul s2. Următorul program, *strspn.c*, ilustrează modul de utilizare a acestei funcții:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    printf("Cauta Abc in AbcDef %d\n", strspn("AbcDef", "Abc"));
    printf("Cauta cbA in AbcDef %d\n", strspn("AbcDef", "cbA"));
    printf("Cauta Def in AbcAbc %d\n", strspn("AbcAbc", "Def"));
  }
```

Atunci când compilați și executați acest program, pe ecran vor apărea următoarele:

```
Cauta Abc in AbcDef 3
Cauta cbA in AbcDef 3
Cauta Def in AbcAbc 0
C:\>
```

Pentru a înțelege mai bine funcția *strspn*, observați următoarea implementare:

```
size_t strspn(const char *s1, const char *s2)
  {
    int i, j;

    for (i = 0; *s1; i++, s1++)
      {
        for (j = 0; s2[j]; j++)
          if (*s1 == s2[j])
            break;
        if (s2[j] == NULL)
          break;
      }
    return(i);
  }
```

191 LOCALIZAREA UNUI SUBȘIR ÎNTR-UN ȘIR DE CARACTERE



Atunci când programele dumneavoastră lucrează cu șiruri de caractere, uneori trebuie să căutați un anumit subșir într-un șir de caractere. Pentru aceasta, cele mai multe compilatoare de C oferă funcția *strstr*, prezentată mai jos:

```
#include <string.h>

char *strstr(sir, subsir);
```

Dacă respectivul subșir este inclus în șir, funcția *strstr.c* returnează un pointer la prima apariție a subșirului, iar dacă nu întâlnește subșirul, returnează *NULL*. Următorul program, *strstr.c*, ilustrează modul de utilizare a acestei funcții:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    printf("Cauta Abc in AbcDef %s\n",
      (strstr("AbcDef", "Abc")) ? "Apare" : "Nu apare");
    printf("Cauta Abc in abcDef %s\n",
      (strstr("abcDef", "Abc")) ? "Apare" : "Nu apare");
    printf("Cauta Abc in AbcAbc %s\n",
      (strstr("AbcAbc", "Abc")) ? "Apare" : "Nu apare");
  }
```

Pentru a vă ajuta să înțelegeți mai bine funcția *strstr*, studiați următoarea implementare:

```
char *strstr(const char *s1, const char *s2)
  {
    int i, j, k;

    for (i = 0; s1[i]; i++)
      for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
        if (! s2[k+1])
          return (s1 + i);
    return(NULL);
  }
```

NUMĂRUL DE APARIȚII ALE UNUI SUBȘIR

C/C++ 192

În secțiunea 191, ați învățat cum să utilizați funcția *strstr* pentru a localiza un subșir într-un șir de caractere. Puteți să cunoașteți de câte ori apare un subșir în cadrul unui șir de caractere. Următoarea funcție, *strstr_nr*, vă permite să determinați de câte ori apare un anumit subșir în cadrul unui șir de caractere:

```
int strstr_nr(const char *sir, const char *subsir)
  {
    int i, j, k, nr = 0;

     for (i = 0; sir[i]; i++)
      for (j = i, k = 0; sir[j] == subsir[k]; j++, k++)
        if (! subsir[k + 1])
          nr++;
    return(nr);
  }
```

OBȚINEREA INDEXULUI UNUI SUBȘIR

C/C++ 193

În secțiunea 191, ați învățat cum să utilizați funcția *strstr* pentru a obține un pointer la un subșir dintr-un șir. Dacă tratați șirurile de caractere ca tablouri, puteți să cunoașteți indexul

caracterului de unde începe respectivul subșir în șirul de caractere. Utilizând valoarea pe care o returnează *strstr*, puteți scădea adresa șirului de caractere pentru a obține indexul:

```
index = strstr(sir, subsir) - sir;
```

Dacă *strstr* nu întâlnește subșirul, valoarea indexului va fi egală cu lungimea șirului. În plus, programele dumneavoastră pot obține indexul unui subșir cu funcția *substr_index*, cum se vede mai jos:

```
int substr_index(const char *s1, const char *s2)
  {
    int i, j, k;

    for (i = 0; s1[i]; i++)
      for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
        if (! s2[k + 1])
          return(i);
           return(i);
  }
```

194 OBȚINEREA ULTIMEI APARIȚII A UNUI SUBȘIR

C/C++

În secțiunea 191 ați utilizat funcția *strstr* pentru a determina prima apariție a unui subșir într-un șir de caractere. În funcție de scopul programului dumneavoastră, puteți să cunoașteți ultima (cea mai din dreapta) apariție a unui subșir într-un șir de caractere. Următoarea funcție, *dr_strstr*, returnează un pointer la ultima apariție a unui subșir într-un șir de caractere sau valoarea *NULL* dacă subșirul nu este inclus:

```
char *dr_strstr(const char *s1, const char *s2)
  {
    int i, j, k, st = 0;

    for (i = 0; s1[i]; i++)
      for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
        if (! s2[k + 1])
          st = i;

    return((st) ? s1+st : NULL);
  }
```

195 AFIȘAREA UNUI ȘIR FĂRĂ SPECIFICATORUL DE FORMAT %s

C/C++

Câteva din secțiunile acestui capitol au utilizat specificatorul de format *%s* pentru a afișa șiruri de caractere. Următoarea instrucțiune, de exemplu, utilizează *printf* pentru a afișa conținutul unei variabile de tip șir de caractere, al cărei nume este *titlu*:

```
printf("%s", titlu);
```

Primul argument dat instrucțiunii *printf* este un șir de caractere care poate conține unul sau mai mulți specificatori de format. Atunci când programele utilizează *printf* pentru a afișa numai un șir de caractere, ca în exemplul precedent, puteți să omiteți șirul de caractere care conține specificatorul de format și să transmiteți ca argument al funcției *printf* numai șirul pe care vreți să-l afișați, ca mai jos:

```
printf(titlu);
```

Așa cum puteți vedea, primul argument al funcției *printf* nu este decât un șir de caractere care conține unul sau mai multe simboluri speciale.

ȘTERGEREA UNUI SUBȘIR DINTR-UN ȘIR DE CARACTERE

C/C++196

În secțiunea 191, ați utilizat funcția *strstr* pentru a determina locația de început a unui subșir într-un șir de caractere. În multe cazuri, programul dumneavoastră trebuie să șteargă un anumit subșir care apare într-un șir de caractere. Pentru a face aceasta, puteți utiliza funcția *strstr_rem*, care înlătură prima apariție a unui subșir, cum se vede mai jos:

```
char *strstr_rem(const char *sir, const char *subsir)
  {
    int i, j, k, loc = -1;

    for (i = 0; sir[i] && (loc == -1); i++)
      for (j = i, k = 0; sir[j] == subsir[k]; j++, k++)
        if (! subsir[k + 1])
          loc = i;
    if (loc != -1) // Subsirul a fost gasit
      {
        for (k = 0; subsir[k]; k++)
          ;
        for (j = loc, i = loc + k, sir[i]; j++, i++)
          sir[j] = sir[i];
        sir[i] = NULL;
      }
    return(sir);
  }
```

ÎNLOCUIREA UNUI SUBȘIR CU ALTUL

C/C++197

În secțiunea 196, ați utilizat funcția *strstr_rem* pentru a înlătura un subșir dintr-un șir de caractere. În multe cazuri, programele dumneavoastră trebuie să înlocuiască prima apariție a unui subșir cu un alt subșir. Puteți face aceasta cu funcția *strstr_rep*, prezentată mai jos:

```
#include <string.h>

char *strstr_rep(char *sursa, char *vechi, char *nou)
  {
```

```
    char *initial = sursa;
    char temp[256];
    int lung_veche = strlen(vechi);
    int i, j, k, loc = -1;

    for (i = 0; sursa[i] && (loc == -1); ++i)
      for (j = i, k = 0; sursa[j] == vechi[k]; j++, k++)
        if (! vechi[k+1])
          loc = i;
    if (loc != -1)
      {
        for (j=0; j<loc; j++)
          temp[j] = sursa[j];
        for (i=0; nou[i]; i++, j++)
          temp[j] = nou[i];
        for (k = loc + lung_veche; sursa[k]; k++, j++)
          temp[j] = sursa[k];
        temp[j] = NULL;
        for (i = 0; sursa[i] == temp[i]; i++) ; // Bucla vida
      }
    return(initial);
  }
```

198 CONVERTIREA UNEI REPREZENTĂRI ASCII NUMERICE

C/C++

Atunci când programele dumneavoastră lucrează cu șiruri de caractere, adesea ele trebuie să convertească o reprezentare ASCII a unei valori, cum ar fi 1.2345, într-o valoare corespunzătoare *int, float, double, long* sau *unsigned*. Pentru a vă ajuta să efectuați această operație, limbajul C dispune de funcțiile prezentate în tabelul 198.

Funcție	Utilitate
atof	Convertește o reprezentare ASCII a unei valori reale în virgulă mobilă
atoi	Convertește o reprezentare ASCII a unei valori întregi
atol	Convertește o reprezentare ASCII a unei valori întregi de tip long

Tabelul 198 *Funcțiile limbajului C pe care le puteți folosi pentru conversia reprezentărilor ASCII în numere.*

Formatul funcțiilor din tabelul 198 este următorul:

```
#include <stdlib.h>

double atof(char *sir);
int atoi(char *sir);
int atol(char *sir);
```

Dacă o funcție nu poate să convertească șirul de caractere într-o valoare numerică, funcția va returna 0. Următorul program, *ascii_to.c*, ilustrează utilizarea funcțiilor *ato*:

```
#include <stdio.h>
#include <stdlib.h>

void main (void)
 {
   int val_int;
   float val_flt;
   long val_long;

   val_int = atoi("12345");
   val_flt = atof("33.45");
   val_long = atol("12prost");
   printf("int %d float %5.2f long %ld\n", val_int, val_flt,
     val_long);
 }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră vor apărea următoarele:

```
int 12345 float 33.45 long 12
C:\>
```

Observați apelul funcției *atol*. Așa cum puteți vedea, atunci când funcția întâlnește valoarea nenumerică (litera p), funcția încheie conversia, returnând valoarea pe care funcția a convertit-o deja până în acel punct.

TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE ALFANUMERIC

C/C++ 199

Un caracter *alfanumeric* este ori o literă, ori o cifră. Cu alte cuvinte, un caracter alfanumeric este ori o majusculă cuprinsă între A și Z, ori o literă mică cuprinsă între a și z, ori o cifră cuprinsă între 0 și 9. Pentru a ajuta programele dumneavoastră să determine dacă un caracter este alfanumeric, fișierul antet *ctype.h* conține o macroinstrucțiune numită *isalnum*. Macroinstrucțiunea examinează un caracter și returnează valoarea 0 dacă nu este alfanumeric sau o valoare diferită de 0 pentru caractere alfanumerice, ca mai jos:

```
if (isalnum(caracter))
```

Pentru a înțelege mai bine macroinstrucțiunea *isalnum*, studiați următoarea implementare:

```
#define isalnum(c) ((toupper((c)) >= 'A' && \ (toupper((c))
<= 'Z')||((c) >= '0' && ((c) <= '9'))
```

200 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE LITERĂ

C/C++

Când programele dumneavoastră lucrează cu şiruri de caractere, va trebui uneori să testaţi dacă un caracter conţine o literă din alfabet (majusculă sau minusculă). Pentru a ajuta programele dumneavoastră să determine dacă un caracter este o literă a alfabetului, fişierul antet *ctype.h* vă pune la dispoziţie macroinstrucţiunea *isalpha*. Macroinstrucţiunea examinează un caracter şi returnează valoarea 0 dacă nu este o majusculă de la A la Z sau o literă mică de la a la z. Dacă este o literă a alfabetului, macroinstrucţiunea returnează o valoare diferită de 0:

```
if (isalpha(caracter))
```

Pentru a înţelege mai bine macroinstrucţiunea *isalpha*, studiaţi următoarea implementare:

```
#define isalpha(c) (toupper((c)) >= 'A' && \ (toupper((c)) <= 'Z')
```

201 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE O VALOARE ASCII

C/C++

O valoare ASCII este o valoare care aparţine intervalului de la 0 la 127. Atunci când programele dumneavoastră lucrează cu şiruri de caractere, pot apărea situaţii în care aveţi nevoie să determinaţi dacă un caracter este o valoare ASCII. Pentru aceasta, fişierul antet *ctype.h* vă pune la dispoziţie macroinstrucţiunea *isascii*, care examinează un caracter şi returnează valoarea 0 dacă este un caracter ASCII sau o valoare diferită de 0 dacă nu este caracter ASCII, aşa cum se vede mai jos:

```
if (isascii(caracter))
```

Pentru a înţelege mai bine macroinstrucţiunea *isascii*, studiaţi următoarea implementare:

```
#define isascii(ltr) ((unsigned) (ltr) < 128)
```

După cum puteţi vedea, macroinstrucţiunea *isascii* consideră o valoare de la 0 la 127 ca valoare ASCII.

202 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE CARACTER DE CONTROL

C/C++

Un *caracter de control* este o valoare de la ^A la ^Z sau de la ^a la ^z. Aplicaţiile folosesec în mod diferit caracterele de control. De exemplu, mediul DOS foloseşte caracterul CTRL+Z pentru a reprezenta sfârşitul unui fişier. Unele procesoare de text utilizează caracterele de control pentru a reprezenta caracterele aldine sau italice. Atunci când lucraţi cu şiruri de caractere, pot apărea situaţii în care trebuie să determinaţi dacă un caracter este caracter de control. Pentru aceasta, fişierul antet *ctype.h* conţine macroinstrucţiunea *iscntrl*, care returnează o valoare diferită de 0 dacă este un caracter de control sau valoarea 0 în caz contrar, aşa cum se vede mai jos:

```
if (iscntrl(caracter))
```

TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE O CIFRĂ

C/C++203

O cifră este o valoare ASCII de la 0 la 9. Atunci când lucrați cu șiruri de caractere, pot apărea situații în care trebuie să stabiliți dacă un caracter este o cifră. Pentru aceasta, fișierul antet *ctype.h* conține macroinstrucțiunea *isdigit*, care examinează caracterul și returnează valoarea 0 dacă nu este o cifră sau o valoare diferită de 0 pentru caracterele din intervalul de la 0 la 9, așa cum se vede mai jos:

```
if (isdigit(litera))
```

Pentru a înțelege mai bine macroinstrucțiunea *isdigit*, observați următoarea implementare:

```
#define isdigit(c) ((c) >= '0' && (c) <= '9')
```

TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE CARACTER GRAFIC

C/C++204

Un *caracter grafic* este un caracter ce poate fi tipărit (vezi *isprint*), cu excepția caracterului spațiu (ASCII 32). Atunci când programele dumneavoastră execută operații care produc caractere, pot apărea situații în care doriți să știți dacă acestea pot fi tipărite sau nu. Pentru a ajuta programele dumneavoastră să efectueze acest test, fișierul antet *ctype.h* pune la dispoziție macroinstrucțiunea *isgraph*, care examinează caracterul și returnează valoarea 0 dacă respectivul caracter nu este grafic sau o valoare diferită de 0 pentru caracterele grafice:

```
if (isgraph(litera))
```

Pentru a înțelege mai bine macroinstrucțiunea *isgraph*, studiați următoarea implementare:

```
#define isgraph(ltr) ((ltr) >= 33) && ((ltr) <= 127)
```

După cum puteți vedea, un caracter grafic este orice caracter ASCII din intervalul de la 33 la 127.

TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE MAJUSCULĂ SAU MINUSCULĂ

C/C++205

Când programele dumneavoastră lucrează cu șiruri de caractere, pot apărea situații în care trebuie să stabiliți dacă un caracter este literă mare sau literă mică. Pentru a ajuta programele dumneavoastră să efectueze acest test, fișierul antet *ctype.h* pune la dispoziție macroinstrucțiunile *islower* și *isupper*. Aceste macroinstrucțiuni examinează caracterul și returnează valoarea 0 dacă nu este minusculă (*islower*) sau majusculă (*isupper*), iar în cazurile contrare întorc o valoare diferită de 0:

```
if (islower(caracter))
if (isupper(caracter))
```

Pentru a înțelege mai bine macroinstrucțiunile *islower* și *isupper*, analizați următoarea implementare:

```
#define islower(c) ((c) >= 'a' && (c) <= 'z')
#define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

206 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ POATE FI TIPĂRIT

C/C++

Atunci când programele dumneavoastră produc ieşiri sub formă de caractere, pot apărea situaţii în care trebuie să testaţi fiecare caracter pentru a vă asigura că poate fi tipărit. Un caracter care poate fi tipărit este orice caracter din intervalul de la 32 (caracterul spaţiu) la 127 (caracterul Del). Pentru a ajuta programele dumneavoastră să testeze dacă un caracter poate fi tipărit, fişierul antet *ctype.h* oferă macroinstrucţiunea *isprint*, care returnează o valoare diferită de 0 pentru caractere ce pot fi tipărite sau valoarea 0 pentru caracterele care nu pot fi tipărite:

```
if (isprint(caracter))
```

Pentru a înţelege mai bine macroinstrucţiunea *isprint*, studiaţi următoarea implementare:

```
#define isprint(ltr) ((ltr) >= 32) && ((ltr) <= 127)
```

După cum puteţi vedea, macroinstrucţiunea *isprint* consideră drept caracter ce poate fi tipărit orice caracter ASCII din intervalul de la 32 la 127.

207 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE SEMN DE PUNCTUAŢIE

C/C++

În cărţi, semnele de punctuaţie sunt virgula, punctul şi virgula, punctul, semnul întrebării şi aşa mai departe. În schimb, limbajul C consideră semn de punctuaţie orice caracter grafic ASCII care nu este alfanumeric. Când programele dumneavoastră lucrează cu şiruri de caractere, pot apărea situaţii în care trebuie să examinaţi dacă un caracter este un semn de punctuaţie. Pentru aceasta, fişierul antet *ctype.h* pune la dispoziţie macroinstrucţiunea *ispunct*, care analizează un caracter şi returnează o valoare diferită de 0 dacă nu este semn de punctuaţie sau valoarea 0 în caz contrar:

```
if (ispunct(caracter))
```

Pentru a înţelege mai bine macroinstrucţiunea *ispunct*, studiaţi următoarea implementare:

```
#define ispunct(c) (isgraph(c)) && ! (isalnum(c))
```

208 TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE SPAŢIU ALB

C/C++

Temenul *spaţiu alb* se referă la următoarele caractere: spaţiu, tabulare, retur de car, linie nouă, tabulare verticală şi avans hârtie. Atunci când programele dumneavoastră creează ieşiri sub formă de caractere, trebuie uneori să testaţi dacă un caracter este sau nu un spaţiu alb. Pentru aceasta, fişierul antet *ctype.h* furnizează macroinstrucţiunea *isspace*, care anali-

zează un caracter și returnează o valoare diferită de 0 dacă este spațiu alb sau valoarea 0 în caz contrar:

```
if (isspace(caracter))
```

Pentru a înțelege mai bine macroinstrucțiunea *isspace*, studiați următoarea implementare:

```
#define isspace(c) (((c) == 32) || ((c) == 9) || ((c) == 13))
```

TESTAREA UNUI CARACTER PENTRU A STABILI DACĂ ESTE O VALOARE HEXAZECIMALĂ

C/C++ 209

O *valoare hexazecimală* este o cifră din intervalul de la 0 la 9, o literă majusculă de la A la Z sau o literă minusculă de la a la z. Atunci când programele dumneavoastră lucrează cu șiruri de caractere, uneori va trebui să determinați dacă un caracter este o cifră hexazecimală. Pentru aceasta, fișierul antet *ctype.h* furnizează macroinstrucțiunea *isxdigit*, care analizează un caracter și returnează o valoare diferită de 0 dacă este o valoare hexazecimală sau valoarea 0 în caz contrar:

```
if (isxdigit(caracter))
```

Pentru a înțelege mai bine macroinstrucțiunea *isxdigit*, studiați următoarea implementare:

```
#define isxdigit(c) (isnum((c)) || (toupper((c)) >= 'A' &&
  toupper((c)) <= 'F'))
```

CONVERTIREA UNUI CARACTER ÎN MAJUSCULĂ

C/C++ 210

Când lucrați cu șiruri de caractere, una dintre operațiile pe care programele dumneavoastră trebuie să le execute de obicei este convertirea unui caracter din minusculă în majusculă. Atunci când vreți să faceți această conversie, aveți două posibilități. Programul poate să folosească macroinstrucțiunea *_toupper*, care este definită în fișierul antet *ctype.h*, sau poate să utilizeze funcția run-time de bibliotecă *toupper*. Macroinstrucțiunea și funcția de bibliotecă au următorul format:

```
#include <ctype.h>
int _toupper(int caracter);
int toupper(int caracter);
```

Deși atât macroinstrucțiunea, cât și funcția de bibliotecă convertesc caracterele în majuscule, ele lucrează diferit. Macroinstrucțiunea *_toupper* nu testează dacă va converti o minusculă. Dacă invocați macroinstrucțiunea pentru un caracter care nu este minusculă, aceasta va genera o eroare. Funcția *toupper*, pe de altă parte, convertește numai literele mici și lasă toate celelalte caractere neschimbate. Dacă sunteți sigur că litera pe care vreți să o convertiți este minusculă, folosiți macroinstrucțiunea *_toupper*, deoarece se execută mai rapid decât funcția. Dacă însă nu sunteți sigur că aveți un caracter minuscul, folosiți funcția *toupper*. Următorul program, *toupper.c*, ilustrează utilizarea macroinstrucțiunii *_toupper* și a funcției de bibliotecă *toupper*, precum și erorile care pot apărea când folosiți macroinstrucțiunea pentru caractere care nu sunt minuscule:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
   {
     char sir[] = "Totul despre C/C++";
     int i;

     for (i = 0; sir[i]; i++)
       putchar(toupper(sir[i]));
     putchar('\n');
     for (i = 0; sir[i]; i++)
       putchar(_toupper(sir[i]));
     putchar('\n');
   }
```

Atunci când compilaţi şi executaţi acest program, pe ecranul dumneavoastră va fi afişat primul şir de caractere corect, cu litere mari (folosind *toupper*). Al doilea şir de caractere va conţine însă caractere non-standard (simboluri, desene şi altele) pentru că *_toupper* încearcă să convertească în majuscule caractere care nu sunt litere mici.

211 CONVERTIREA UNUI CARACTER ÎN MINUSCULĂ

C/C++

Când lucraţi cu şiruri de caractere, una dintre operaţiile pe care programele dumneavoastră trebuie să le execute de obicei este convertirea unui caracter din majusculă în minusculă. Atunci când vreţi să faceţi această conversie, aveţi două posibilităţi. Programul dumneavoastră poate folosi macroinstrucţiunea *_tolower*, care este definită în fişierul antet *ctype.h*, sau poate utiliza funcţia de bibliotecă run-time *tolower*. Macroinstrucţiunea şi funcţia de bibliotecă au următorul format:

```
#include <ctype.h>

int _tolower(int caracter);
int tolower(int caracter);
```

Deşi atât macroinstrucţiunea, cât şi funcţia de bibliotecă convertesc caracterele la majusculă, ele lucrează diferit. Macroinstrucţiunea *_tolower* nu testează dacă va converti o majusculă. Dacă invocaţi macroinstrucţiunea pentru un caracter care nu este majusculă, se va genera o eroare. Funcţia *tolower*, pe de altă parte, converteşte numai majusculele şi lasă toate celelalte caractere neschimbate. Dacă sunteţi sigur că litera pe care vreţi să o convertiţi este majusculă, folosiţi macroinstrucţiunea *_tolower*, deoarece se execută mai rapid decât funcţia. Dacă însă nu sunteţi sigur că respectivul caracter este majusculă, folosiţi funcţia *tolower*. Următorul program, *tolower.c*, ilustrează utilizarea macroinstrucţiunii *_tolower* şi a funcţiei *tolower*, precum şi erorile care pot apărea atunci când folosiţi macroinstrucţiunea pentru caractere care nu sunt majuscule:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
   {
```

```
    char sir[] = "Totul despre C/C++";
    int i;

    for (i = 0; sir[i]; i++)
      putchar(tolower(sir[i]));
    putchar('\n');
    for (i = 0; sir[i]; i++)
      putchar(_tolower(sir[i]));
    putchar('\n');
 }
```

Atunci când compilați și executați acest program, pe ecranul dumneavoastră va fi afișat primul șir de caractere corect, cu litere mici (folosind *tolower*). Al doilea șir de caractere va conține însă caractere non-standard (simboluri, grafice și altele) pentru că *_tolower* încearcă să convertească în litere mici caractere care nu sunt litere mari.

LUCRUL CU CARACTERE ASCII

C/C++ 212

Atunci când lucrați cu șiruri de caractere și cu funcții de prelucrare a caracterelor, trebuie să vă asigurați uneori că un caracter este un caracter ASCII valid, ceea ce înseamnă că valoarea este în intervalul de la 0 la 127. Pentru a vă asigura că un caracter este caracter ASCII valid, puteți folosi macroinstrucțiunea *toascii*, care este definită în fișierul antet *ctype.h*, ca mai jos:

```
#include <ctype.h>

int toascii(int caracter);
```

Pentru a înțelege mai bine macroinstrucțiunea *toascii*, studiați următoarea implementare:

```
#define toascii(caracter) ((caracter) & 0x7F)
```

Pentru a fi mai performantă, macroinstrucțiunea *toascii* execută operația *ȘI pe biți*, care șterge cel mai semnificativ bit din octetul care reprezintă caracterul. Operația *ȘI pe biți* asigură plasarea valorilor în intervalul 0-127.

SCRIEREA IEȘIRII FORMATATE ÎNTR-O VARIABILĂ DE TIP ȘIR DE CARACTERE

C/C++ 213

După cum știți, funcția *printf* vă permite să formatați ieșirea către ecranul monitorului. În funcție de cerințele programelor dumneavoastră, pot apărea situații în care trebuie să lucrați cu șiruri de caractere care conțin o ieșire formatată. De exemplu, să presupunem că angajații dumneavoastră au un număr de marcă compus din cinci cifre și un identificator de regiune din trei caractere (cum ar fi Sea pentru Seattle). Să presupunem că păstrați informațiile despre fiecare angajat într-un fișier pe care îl numiți cu o combinație a celor două valori (cum ar fi SEA12345). Funcția *sprintf* vă permite să formatați ieșirea într-un șir de caractere. Formatul funcției *sprintf* este următorul:

```
#include <stdio.h>

int sprintf(char *sir,const char *format [,argumente...]);
```

Următorul program, *sprintf.c*, utilizează funcția *sprintf* pentru a crea numele de fișier al unui angajat:

```
#include <stdio.h>

void main(void)
  {
    int numar_marca = 12345;
    char reg[] = "SEA";
    char numefis[64];

    sprintf(numefis, "%s%d", reg, numar_marca);
    printf("Fisierul angajatului: %s\n", numefis);
  }
```

214 CITIREA INTRĂRII DINTR-UN ȘIR DE CARACTERE C/C++

Așa cum ați învățat, funcția *scanf* vă permite să citiți o intrare furnizată de *stdin*. În funcție de programele dumneavoastră, uneori se va întâmpla ca un șir de caractere să conțină câmpuri pe care doriți să le atribuiți anumitor variabile. Funcția *sscanf* permite programelor dumneavoastră să citească valori dintr-un șir de caractere și să le atribuie variabilelor specificate. Formatul funcției *sscanf* este următorul:

```
#include <stdio.h>

int sscanf(const char *sir, const char *format [, argumente]);
```

Argumentele pe care programul dumneavoastră le transmite funcției *sscanf* trebuie să fie pointeri la variabile de tip adresă. Dacă *sscanf* atribuie câmpurile cu succes, returnează numărul de câmpuri atribuite. Dacă *sscanf* nu atribuie câmpuri, atunci returnează 0 sau EOF în cazul în care întâlnește un terminator de șir. Următorul program, *sscanf.c*, ilustrează utilizarea funcției *sscanf*:

```
#include <stdio.h>

void main(void)
  {
    int varsta;
    float salariu;
    char sir[] = "33 2500000.0";

    sscanf(sir, "%d %f\n", &varsta, &salariu);
    printf("Varsta: %d Salariu %f\n", varsta, salariu);
  }
```

215 SIMBOLIZAREA ȘIRURILOR DE CARACTERE PENTRU A ECONOMISI SPAȚIU C/C++

Simbolizarea șirurilor de caractere este procesul de utilizare a unei valori unice pentru a reprezenta un șir. De exemplu, să presupunem că aveți un program care lucrează cu un

număr mare de caractere. Să spunem că programul conține o bază de date cu conturile clienților dumneavoastră pe orașe și state. În funcție de modul în care este conceput programul, puteți ajunge la un volum mare de teste, așa cum se vede mai jos:

```
if (streql(oras, "Seattle"))
   // Instructiune
else if (streql(oras, "New York"))
   // Instructiune
else if (streql(oras, "Chicago"))
   // Instructiune
```

În cadrul fiecărui fragment de program care execută în mod repetat teste *else if*, se consumă un spațiu considerabil pentru stocarea constantelor de tip șir de caractere și mult timp pentru compararea șirurilor. În loc să utilizăm apeluri repetate la șiruri, putem crea o funcție denumită *tokenize_string* care să returneze un simbol unic pentru fiecare șir de caractere. În cadrul funcției din exemplul precedent, testarea din program va arăta astfel:

```
int simbol_oras;

simbol_oras = tokenize_string(oras);
if (simbol_oras == simbol_Seattle)
   // Instructiune
else if (simbol_oras == simbol_NewYork)
   // Instructiune
else if (simbol_oras == simbol_Chicago)
   // Instructiune
```

Utilizând simbolurile în modul descris mai sus, veți putea elimina spațiul consumat de constantele de tip șir de caractere. De asemenea, eliminarea comparației dintre șiruri va duce la îmbunătățirea performanțelor programului dumneavoastră.

INIȚIALIZAREA UNUI ȘIR DE CARACTERE

C/C++ 216

În capitolul „Matrice, pointeri și structuri", veți afla cum să atribuiți valori unei matrice în timp ce este declarată de program. Limbajul C reprezintă șirul de caractere sub forma unei matrice (tablou) de octeți. Atunci când declarați un șir, veți specifica, în general, o valoare inițială, ca mai jos:

```
char titlu[] = "Totul despre C/C++";
char sectiune[64] = "Siruri";
```

Compilatorul de C va atribui un tablou suficient de mare pentru stocarea caracterelor specificate în dreptul șirului *titlu* (precum și a caracterului *NULL*). Pentru că șirul Totul despre C/C++ conține 35 de caractere, șirul *titlu* poate păstra 35 de caractere ce pot fi tipărite plus caracterul *NULL*. Dacă, mai târziu, veți atribui mai mult de 32 de caractere șirului, veți suprascrie o zonă de memorie care stochează valoarea altei variabile. Pentru șirul *sectiune*, compilatorul va aloca un spațiu capabil să stocheze 64 de caractere. Compilatorul va atribui primilor 8 octeți ai șirului literele cuvântului „Șiruri" și celui de al noulea, caracterul *NULL*. De obicei, ceilalți 55 de octeți vor fi inițializați cu caracterul *NULL*.

217 PREZENTAREA FUNCȚIILOR



Cele mai multe dintre programele prezentate până acum în această carte au utilizat numai funcția *main*. Pe măsură ce programele dumneavoastră vor deveni mai mari și mai complexe, veți putea să vă simplificați munca și să îmbunătățiți claritatea acestora prin fragmentarea lor în părți mai mici, numite *funcții*. De exemplu, să presupunem că ați creat un program de contabilitate. Puteți să aveți o funcție care să efectueze operațiile din registre, altă funcție pentru debite, o a treia funcție pentru credite și o a patra pentru generarea balanței de plăți. Dacă ați plasa toate instrucțiunile în cadrul funcției *main*, programul ar deveni foarte lung și greu de înțeles. Pe măsură ce dimensiunea programelor și complexitatea lor crește, crește și posibilitatea apariției erorilor. Dacă fragmentați programul în părți mai mici, mai ușor de manevrat, puteți evita erorile. O *funcție* este o colecție de instrucțiuni care execută o anumită sarcină. De exemplu, următoarea funcție, *salut_planeta*, folosește funcția *printf* pentru a afișa un mesaj:

```
void salut_planeta(void)
  {
    printf("Salut, planeta!\n");
  }
```

Cuvântul cheie *void* spune compilatorului de C că funcția nu returnează o valoare. De multe ori, funcțiile folosesc instrucțiunea *return* pentru a returna rezultatul calculelor către funcția care le apelează. Dacă funcția nu folosește *return* pentru returnarea rezultatului, trebuie să precedați numele funcției cu *void*. Cuvântul *void*, care apare între paranteze, spune compilatorului de C că funcția nu utilizează nici un parametru. Un *parametru* este o informație pe care programul o transmite funcției. Când programele apelează funcția *printf*, de exemplu, informațiile plasate între paranteze sunt *parametri*. Când o funcție nu folosește parametri, trebuie să plasați între paranteze cuvântul *void*. Pentru a utiliza o funcție, trebuie pur și simplu să specificați numele funcției urmat de paranteze, așa cum ați folosit funcția *printf*. Programatorii denumesc utilizarea unei funcții *apel de funcție*. Următorul program, *utilfunc.c*, utilizează funcția *salut_planeta*:

```
#include <stdio.h>
void salut_planeta(void)
  {
    printf("Salut, planeta!\n");
  }
void main(void)
  {
    salut_planeta();
  }
```

Când rulați acest program, întâi se execută funcția *main*. După cum puteți vedea, unica instrucțiune din *main* este apelul funcției denumite *salut_planeta*. Când compilatorul întâlnește apelul de funcție, el transferă imediat execuția programului către funcție și se începe cu prima instrucțiune a acesteia. După terminarea ultimei instrucțiuni din funcție, compilatorul de C transferă execuția către instrucțiunea care urmează imediat după apelul funcției. Pentru a înțelege mai bine acest proces, să modificăm funcția *main* din cadrul programului *utilfunc.c*, ca mai jos:

```
void main(void)
  {
    printf("Incepe apelarea functiei\n");
    salut_planeta();
    printf("S-a terminat apelarea functiei\n");
  }
```

Atunci când compilați și executați programul *utilfunc.c*, pe ecranul dumneavoastră vor fi afișate următoarele:

```
Incepe apelarea functiei
Salut, planeta!
S-a terminat apelarea functiei
C:\>
```

UTILIZAREA VARIABILELOR ÎN CADRUL FUNCȚIILOR

C/C++ 218

Pe măsură ce veți crea diferite funcții, veți observa că multe necesită variabile pentru a genera rezultatele dorite. Când folosiți o variabilă în cadrul unei funcții, mai întâi trebuie să o declarați, la fel ca în cazul funcției *main*. De exemplu, următorul program, *trei_sal.c*, apelează funcția *trei_saluturi*, care folosește variabila *contor* în bucla *for* pentru a afișa un mesaj de trei ori:

```
#include <stdio.h>

void trei_saluturi(void)
  {
    int contor;        // Variabila

    for (contor = 1; contor <= 3; contor++)
      printf("Salut, planeta!\n");
  }

void main(void)
  {
    trei_saluturi();
  }
```

Când declarați variabile în cadrul funcțiilor, numele pe care le dați acestora se folosesc exclusiv în cadrul funcției. De aceea, chiar dacă programul dumneavoastră utilizează 10 funcții diferite și fiecare folosește o variabilă numită *contor*, compilatorul de C va considera diferită variabila fiecărei funcții. Dacă funcția dumneavoastră necesită mai multe variabile, trebuie să le declarați la începutul funcției, la fel ca în cadrul funcției *main*.

FUNCȚIA MAIN

C/C++ 219

Atunci când creați un program în C, numele de funcție *main* determină prima instrucțiune pe care o va executa programul. De fapt, *main* este o funcție, astfel încât dacă aveți întrebări în legătură cu tipurile de operații pe care le puteți efectua în cadrul funcțiilor dumneavoastră,

regula este relativ simplă: *Orice puteți face în main, puteți face și într-o funcție.* La fel cum puteți declara variabile în *main*, puteți să le declarați și în funcțiile dumneavoastră. Puteți, de asemenea, să folosiți structuri cum ar fi *if*, *while* și *for*. În sfârșit, o funcție o poate apela (folosi) pe alta. De exemplu, următorul program, *apel_2.c*, utilizează două funcții. Când programul începe, funcția *main* apelează funcția *trei_saluturi*, care, la rândul său, apelează de trei ori funcția *salut_planeta* pentru a afișa mesaje pe ecranul dumneavoastră, așa cum se vede mai jos:

```
#include <stdio.h>

void salut_planeta(void)
  {
    printf("Salut, planeta!\n");
  }

void trei_saluturi(void)
  {
    int contor;
    for (contor = 1; contor <= 3; contor++)
      salut_planeta();
  }

void main(void)
  {
    trei_saluturi();
  }
```

220 PREZENTAREA PARAMETRILOR

C/C++

Un parametru este o valoare transmisă unei funcții. Cele mai multe dintre programele prezentate în această carte au transmis parametri către funcția *printf*, ca mai jos:

```
printf("Valoarea este %d\n", rezultat);
```

Atunci când utilizați funcții în mod regulat, puteți să le îmbunătățiți utilitatea transmițându-le parametri. De exemplu, să analizăm următoarea structură a funcției *trei_saluturi*, care apelează de trei ori funcția *salut_planeta*:

```
void trei_saluturi(void)
  {
    int contor;

    for (contor = 1; contor <= 3; contor++)
      salut_planeta();
  }
```

Funcția ar fi și mai utilă dacă v-ar permite să specificați ca parametru numărul care arată de câte ori doriți să afișați mesajul. Pentru a utiliza un parametru, funcția trebuie să specifice numele parametrului și tipul său, ca mai jos:

```
void nr_saluturi(int nr_mesaje)
```

În acest caz, funcția *nr_saluturi* acceptă un parametru de tip *int*, numit *nr_mesaje*. Când o altă funcție, cum ar fi *main*, vrea să folosească *nr_saluturi*, trebuie să specifice valoarea atribuită parametrului *nr_mesaje*:

```
nr_saluturi(2);          // Afiseaza mesajul de doua ori
nr_saluturi(100);        // Afiseaza mesajul de o suta de ori
nr_saluturi(1);          // Afiseaza mesajul o data
```

Următorul program, *uzparam.c*, ilustrează modul în care puteți utiliza o funcție cu parametri:

```
#include <stdio.h>
void salut_planeta(void)
  {
    printf("Salut, planeta!\n");
  }
void nr_saluturi(int nr_mesaje)
  {
    int contor;
    for (contor = 1; contor <= nr_mesaje; contor++)
      salut_planeta();
  }
void main(void)
  {
    printf("Afiseaza mesajul de doua ori\n");
    nr_saluturi(2);
    printf("Afiseaza mesajul de cinci ori \n");
    nr_saluturi(5);
  }
```

După cum vedeți, în *main*, apelul funcției *nr_saluturi* include valoarea pe care compilatorul de C trebuie să o atribuie parametrului *nr_mesaje*.

***Observație:** Când transmiteți un parametru unei funcții, tipul valorii pe care o atribuiți acestuia (cum ar fi **int, float, char** și așa mai departe) trebuie să coincidă cu tipul său. În funcție de compilatorul de C cu care lucrați, acesta poate să detecteze nepotrivirile cu tipul parametrului. Dacă nu sunt detectate aceste nepotriviri, pot apărea erori care adesea sunt dificil de găsit și de corectat.*

UTILIZAREA PARAMETRILOR MULTIPLI

C/C++221

Așa cum ați învățat, un parametru este o valoare pe care o transmiteți funcției. În general, puteți transmite un număr nelimitat de parametri către o funcție. Totuși, cercetările arată că funcția este mai dificil de înțeles și de utilizat corect atunci când numărul de parametri este mai mare de șapte și, de aceea, crește probabilitatea apariției erorilor. Când funcțiile dumneavoastră utilizează mai mulți parametri, trebuie să specificați tipul și numele fiecăruia și să îi separați prin virgulă, ca mai jos:

```
void o_functie(int varsta, float salariu, int nr_marca)
  {
    // Instructiunile functiei
  }
```

Atunci când programul dumneavoastră apelează funcția, trebuie să specificați valoarea fiecărui parametru, ca mai jos:

```
o_functie(33, 400000.00, 534);
```

Pe baza acestor specificații, compilatorul de C va atribui valorile parametrilor, cum se arată în figura 221.

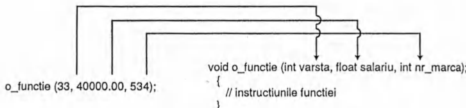


***Figura 221** Schema de atribuire a valorilor parametrilor.*

222 DECLARAȚIILE PARAMETRILOR ÎN PROGRAMELE ÎN C MAI VECHI

C/C++

Atunci când creați funcții care folosesc parametri, de obicei specificați tipurile și numele parametrilor, separate prin virgule, în antetul funcției, ca mai jos:

```
void o_functie(int varsta, float salariu, int nr_marca)
  {
    // Instructiunile functiei
  }
```

Dacă lucrați cu un program în C mai vechi, este posibil ca parametrii să fie declarați astfel:

```
void o_functie(varsta, salariu, nr_marca)
     int varsta;
     float salariu;
     int nr_marca;
  {
    // Instructiunile functiei
  }
```

Dacă întâlniți asemenea declarații de parametri, trebuie să înțelegeți că, deși formatul declarației este puțin diferit, scopul rămâne același – specificarea tipului și numelui parametrilor. Dacă vă simțiți tentat să actualizați formatul funcției, asigurați-vă că noul format este complet acceptat de compilatorul dumneavoastră. De asemenea, amintiți-vă că, pe măsură ce faceți

mai multe modificări programelor dumneavoastră, crește posibilitatea de a introduce o eroare. Ca regulă generală, *nu vă legați la cap dacă nu vă doare!*

RETURNAREA UNEI VALORI DE LA O FUNCȚIE

C/C++ 223

Pe măsură ce programele dumneavoastră vor deveni mai complexe, ele vor efectua de obicei calcule și vor returna un rezultat. Pentru a furniza un rezultat apelantului, funcția trebuie să conțină instrucțiunea *return*, ca mai jos:

```
return(rezultat);
```

Tipul funcției depinde de tipul valorii returnate (*int*, *float*, *char* și așa mai departe). Dacă o funcție returnează valori de tip *int*, de exemplu, trebuie să plasați numele tipului în fața numelui funcției, ca mai jos:

```
int o_functie(int valoare)
  {
    // Instructiunile functiei
  }
```

Următoarea funcție, *i_cub*, returnează cubul unei valori întregi, introduse ca parametru. De exemplu, dacă funcția transmite valoarea 5 funcției apelate, *i_cub* va returna valoarea 5*5*5, adică 125:

```
int i_cub(int valoare)
  {
    return(valoare * valoare * valoare);
  }
```

După cum puteți vedea, funcția folosește instrucțiunea *return* pentru a returna rezultatul calculului către apelant. Funcția apelantă poate atribui rezultatul funcției pe care o apelează (cunoscut sub numele de *valoare de revenire*) unei variabile sau codul poate să folosească valoarea returnată în cadrul unei a treia funcții, cum ar fi *printf*, așa cum se vede mai jos:

```
rezultat = i_cub(5);
printf("Cubul lui 5 este %d\n", i_cub(5));
```

Următorul program, *i_cub.c*, utilizează funcția *i_cub* pentru a determina cuburile mai multor valori:

```
#include <stdio.h>
int i_cub(int val)
  {
    return(val * val * val);
  }
void main(void)
  {
    printf("Cubul lui 3 este %d\n", i_cub(3));
    printf("Cubul lui 5 este %d\n", i_cub(5));
```

```
    printf("Cubul lui 7 este %d\n", i_cub(7));
  }
```

Valorile pe care le transmiteţi funcţiei trebuie să fie de acelaşi tip cu tipul parametrului din declaraţia funcţiei. Dacă doriţi să determinaţi cubul unei valori reale în virgulă mobilă, de exemplu, trebuie să creaţi o a doua funcţie, numită *f_cub*, ca mai jos:

```
float f_cub(float valoare)
  {
    return(valoare * valoare * valoare);
  }
```

224 INSTRUCŢIUNEA RETURN

C/C++

Aşa cum aţi învăţat, pentru a returna un rezultat funcţiei apelante, o funcţie trebuie să folosească instrucţiunea *return*. Când compilatorul de C întâlneşte instrucţiunea *return* într-o funcţie, încheie imediat execuţia funcţiei şi returnează valoarea respectivă funcţiei apelante. Programul nu mai execută eventualele instrucţiuni care urmează instrucţiunii *return* în cadrul funcţiei, ci reia execuţia în funcţia apelantă.

Dacă studiaţi şi alte programe în C, puteţi să întâlniţi funcţii care conţin mai multe instrucţiuni *return*, fiecare din ele returnând o valoare în anumite condiţii. De exemplu, să observăm funcţia *comp_val*, prezentată mai jos:

```
int comp_val(int prima, int a_doua)
  {
   if (prima == a_doua)
     return(0);
   else if (prima > a_doua)
     return(1);
   else if (prima < a_doua)
     return(2);
  }
```

Funcţia *comp_val* examinează două valori întregi şi returnează una dintre valorile prezentate în tabelul 224.

Rezultat	Semnificaţie
0	Valorile sunt aceleaşi.
1	Prima valoare este mai mare decât a doua.
2	A doua valoare este mai mare decât prima.

Tabelul 224 *Valorile returnate de funcţia* ***comp_val***.

Ca regulă, trebuie să încercaţi să vă limitaţi funcţiile la o singură instrucţiune *return*. Pe măsură ce funcţiile devin mai mari şi mai complexe, utilizarea mai multor instrucţiuni *return* le face mai greu de înţeles. În cele mai multe cazuri, puteţi rescrie funcţiile astfel încât să folosească numai o instrucţiune *return*, ca mai jos:

```
int comp_val(int prima, int a_doua)
   {
     int rezutat;
     if (prima == a_doua)
       rezultat = 0;
     else if (prima > a_doua)
       rezultat = 1;
     else if (prima < a_doua)
       rezultat = 2;
     return(rezultat);
   }
```

În acest caz, funcția fiind atât de simplă, va fi greu să înțelegeți avantajul oferit de utilizarea unei singure instrucțiuni *return*. Dar, pe măsură ce funcțiile dumneavoastră vor deveni mai complexe, acest avantaj va deveni din ce în ce mai evident. Trebuie totuși să rețineți că, uneori, codul pe care îl creați este mai lizibil dacă folosiți mai multe instrucțiuni *return*. Trebuie să scrieți un cod cât mai lizibil și mai ușor de modificat; dacă este nevoie de mai multe instrucțiuni *return* pentru atingerea acestui obiectiv, folosiți atâtea câte sunt necesare.

PROTOTIPURILE DE FUNCȚII

C/C++ 225

Dacă observați cu atenție fiecare dintre programele precedente, veți vedea că funcțiile apelante apar întotdeauna în urma funcției pe care o apelează în codul sursă al programului. Cele mai multe dintre noile compilatoare de C cer ca tipul parametrilor și al valorii returnate să fie cunoscute înainte ca programul să apeleze funcția. Plasând funcția în fața celei care o apelează în cadrul codului programului, îi permiteți compilatorului să obțină informațiile necesare înainte de a întâlni apelul de funcție. Pe măsură ce programele dumneavoastră vor deveni mai complexe, uneori va fi imposibil să plasați funcțiile în ordinea corectă. De aceea, limbajul C vă permite să plasați în programul dumneavoastră *prototipuri de funcții*, care precizează tipul parametrilor și al valorilor returnate. De exemplu, să analizăm un program care utilizează funcțiile *i_cub* și *f_cub*, prezentate în secțiunea 223. Înainte de prima utilizare a funcțiilor, poate fi inclus în program un prototip similar cu următorul:

```
int i_cub(int);        // Returneaza un int--un parametru int
float f_cub(float);    // Returneaza un float--un parametru float
```

După cum puteți vedea, prototipul funcției specifică tipul parametrilor și al valorilor returnate. Următorul program, *utilprot.c*, utilizează două prototipuri de funcții pentru a elimina necesitatea ordonării funcțiilor:

```
#include <stdio.h>

int i_cub(int);
float f_cub(float);

void main(void)
   {
     printf("Cubul lui 3 e %d\n", i_cub(3));
     printf("Cubul lui 3.7 e %f\n", f_cub(3.7));
   }
```

```
int i_cub(int val)
  {
    return(val * val * val);
  }
float f_cub(float val)
  {
    return(val * val * val);
  }
```

Dacă studiaţi fişierele antet *.h*, cum ar fi *stdio.h*, veţi observa că aceste fişiere conţin mai multe prototipuri de funcţii.

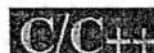
226 BIBLIOTECA RUN-TIME



Când veţi scrie propriile dumneavoastră funcţii, veţi vedea adesea că funcţiile create pentru un program sunt necesare şi în alt program. Abilitatea de a refolosi o funcţie în mai multe programe poate să reducă substanţial timpul de programare şi testare. În secţiunea „Instrumente", veţi învăţa cum să vă plasaţi funcţiile utilizate frecvent într-o bibliotecă pentru a le face mai uşor de folosit în mai multe programe. Pentru moment, puteţi să tăiaţi şi să lipiţi instrucţiunile funcţiilor dintr-un cod sursă în altul.

Înainte de a pierde o mulţime de timp scriind o mare varietate de funcţii pentru diverse necesităţi, este bine să studiaţi funcţiile pe care vi le pune la dispoziţie compilatorul dumneavoastră. În multe compilatoare, aceste funcţii încorporate formează *biblioteca run-time*. De obicei, compilatoarele de C dispun de o bibliotecă run-time cu sute de funcţii, cu scopuri variind de la deschiderea şi manipularea fişierelor şi până la accesarea discului sau a unui director pentru a determina lungimea unui şir de caractere. Veţi pierde o oră sau două citind documentaţia bibliotecii run-time oferite de compilatorul dumneavoastră, dar veţi economisi nenumăratele ore de programare pe care le-aţi petrece „reinventând roata".

227 PARAMETRII FORMALI ŞI REALI



Citind diverse cărţi despre limbajul C, veţi întâlni termenii parametri *formali* şi *reali*. Pe scurt, parametrii formali sunt numele parametrilor care apar în definiţia funcţiei. În exemplul de mai jos, *varsta, salariu* şi *nr_marca* sunt parametri formali pentru funcţia *info_angajat*:

```
void info_angajat(int varsta, float salariu, int nr_marca)
  {
    // Instructiunile functiei
  }
```

Atunci când o funcţie apelează altă funcţie, valorile transmise de funcţia apelantă sunt parametri reali. În cazul următorului apel de funcţie, valorile 30, 420000.00 şi 321 sunt parametrii reali:

```
info_angajat(30, 420000.00, 321);
```

Parametrii reali pe care îi transmiteți unei funcții pot fi valori constante sau variabile. Valoarea sau tipul variabilei trebuie să se potrivească cu parametrul formal. De exemplu, următorul fragment de cod ilustrează modul de utilizare a variabilelor ca parametri reali:

```
int varsta_angajat = 30;
float salariu_angajat = 420000.00;
int nr_marca = 321;

info_angajat(varsta_angajat, salariu_angajat, nr_marca)
```

Atunci când apelați o funcție folosind variabile ca parametri reali, numele variabilelor utilizate nu au nici o legătură cu numele parametrilor formali. Compilatorul de C va lua în considerare numai valorile pe care le au variabilele respective.

REZOLVAREA CONFLICTELOR DE NUME

C/C++ 228

Așa cum ați învățat, cele mai multe compilatoare de C dispun de o vastă bibliotecă de funcții pe care le puteți apela pentru a executa o anumită sarcină. De exemplu, pentru a obține valoarea absolută a unei expresii de tip întreg, puteți folosi funcția *abs*. La fel, pentru a copia conținutul unui șir de caractere în alt șir, puteți folosi funcția *strcpy*. Atunci când creați funcții proprii, unele vor avea același nume ca și funcțiile din biblioteca run-time. De exemplu, următorul program, *mystrcpy.c*, creează și utilizează o funcție numită *strcpy*:

```
#include <stdio.h>

char *strcpy(char *destinatie, const char *sursa)
  {
    char *start = destinatie;

    while (*destinatie++ = *sursa++)
      ;

    return(start);
  }

void main(void)
  {
    char titlu[64];

    strcpy(titlu, "Totul despre C/C++");
    printf(titlu);
  }
```

Când numele unei funcții pe care o declarați în cadrul programelor dumneavoastră intră în conflict cu o funcție din biblioteca run-time, compilatorul de C folosește funcțiile programului dumneavoastră, nu pe cele din biblioteca run-time.

FUNCȚIILE CARE NU RETURNEAZĂ INT

C/C++ 229

Ați văzut mai devreme că multe funcții returnează valori de tip *int*. Când funcția dumneavoastră nu returnează o valoare de tip *int* (ci una de tip *float*, *double*, *char* și așa mai departe), trebuie să indicați compilatorului tipul valorii returnate. Următorul program,

media.c, folosește funcția *val_medie* pentru a determina media a trei valori de tip *int*. Funcția returnează media folosind o valoare de tip *float*:

```
#include <stdio.h>

float val_medie(int a, int b, int c)
  {
    return ((a + b + c) / 3.0);
  }

void main(void)
  {
    printf("Media pentru 100, 133 si 155 este %f\n",
    val_medie(100, 133, 155));
  }
```

După cum puteți vedea, tipul valorii returnate este specificat la începutul funcției:

```
float val_medie(int a, int b, int c)
```

Observație: *Dacă nu specificați tipul valorii returnate, compilatorul de C va presupune că funcția returnează o valoare de tip* ***int.***

230 VARIABILELE LOCALE



Limbajul C vă permite să declarați variabile în cadrul funcțiilor dumneavoastră. Aceste variabile se numesc *variabile locale*, pentru că numele și valorile lor sunt valabile doar în cadrul funcției care conține declarația respectivelor variabile. Următorul program, *local_er.c*, ilustrează conceptul de variabilă locală. Funcția *val_local* declară trei variabile, *a*, *b* și *c* și le atribuie valorile 1, 2 și, respectiv, 3. Funcția *main* încearcă să tipărească valoarea fiecărei variabile. Dar cum numele variabilelor sunt locale funcției *val_local*, compilatorul generează erori care anunță că simbolurile *a*, *b* și *c* sunt nedefinite, așa cum se vede mai jos:

```
#include <stdio.h>

void val_local(void)
  {
    int a = 1, b = 2, c =3;

    printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }

void main(void)
  {
    printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }
```

231 UTILIZAREA STIVEI DE CĂTRE FUNCȚII



Stiva, o zonă pe care programele o utilizează pentru a păstra temporar informații detaliate, este prezentată în capitolul „Memoria". Principala destinație a unei stive este apelarea

funcțiilor. Când programul dumneavoastră apelează o funcție, compilatorul de C plasează în stivă adresa instrucțiunii care urmează apelării funcției (denumită *adresă de revenire*). Apoi, compilatorul de C plasează în stivă parametrii funcției, de la dreapta la stânga. În final, dacă funcția declară variabile locale, compilatorul alocă în stivă un spațiu pe care funcția îl utilizează pentru a stoca valoarea variabilelor respective. Figura 231 arată modul în care compilatorul de C utilizează stiva în cazul unui apel simplu de funcție.

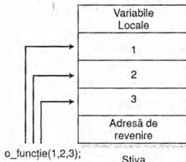


Figura 231 *Compilatorul de C utilizează stiva pentru apelul unei funcții.*

SUPRASARCINA FUNCȚIEI

C/C++ 232

Așa cum ați învățat în secțiunea 231, când programul dumneavoastră folosește funcții, compilatorul de C depune adresa de revenire, parametrii și variabilele locale într-o stivă. Când se termină execuția funcției, compilatorul descarcă zona din stivă care conține variabilele locale și parametrii și apoi folosește valoarea de revenire pentru a relua execuția programului de la locația corectă.

Deși utilizarea stivei este puternică, pentru că permite compilatorului să apeleze funcțiile și să le transmită informații, ea consumă timp de procesare. Intervalul de timp necesar calculatorului pentru a încărca și a descărca informațiile din stivă este numit de programatori – *suprasarcina funcției* (overhead). Pentru a înțelege mai bine impactul suprasarcinii funcției asupra performanțelor programelor dumneavoastră, să studiem următorul program, *funcover.c*. Programul utilizează mai întâi o buclă pentru a însuma valorile de la 1 la 100000. Apoi, utilizează o nouă buclă pentru a aduna valorile, cum se vede mai jos:

```
#include <stdio.h>
#include <time.h>

float aduna(long int a, float b)
  {
    float rezultat;

    rezultat = a + b;
    return(rezultat);
  }

void main(void)
```

```
{
  long int i;
  float rezultat = 0;
  time_t start_time, stop_time;
  printf("Lucreaza...\n");
  time(&start_time);
  for (i = 1; i <= 100000L; i++)
    rezultat += i;
  time(&stop_time);
  printf("Utilizare bucla %d secunde\n", stop_time -
     start_time);
  printf("Lucreaza...\n");
  time(&start_time);
  for (i = 1; i <= 100000L; i++)
    rezultat = aduna(i, rezultat);
  time(&stop_time);
  printf("Utilizare functie %d secunde\n", stop_time -
     start_time);
}
```

În cele mai multe sisteme, calculele bazate pe funcții pot consuma de două ori mai mult timp pentru procesare. De aceea, când folosiți funcții în cadrul programelor dumneavoastră, trebuie să luați în considerare beneficiile pe care le aduce funcția (cum ar fi ușurința utilizării, reutilizarea unei funcții existente, reducerea necesității de testare, ușurința înțelegerii și așa mai departe) față de reducerea performanțelor datorită suprasarcinii introduse.

233 AMPLASAREA VARIABILELOR LOCALE

C/C++

Așa cum ați învățat, limbajul C vă permite să declarați variabile în cadrul funcțiilor dumneavoastră. Aceste variabile sunt *locale* pentru funcția respectivă, ceea ce înseamnă că numai funcția în cadrul căreia ați declarat variabilele recunoaște valorile și existența lor. Următoarea funcție, *util_abc*, declară trei variabile locale, numite *a, b* și *c*:

```
void util_abc(void)
  {
    int a, b, c;
    a = 3;
    b = a + 1;
    c = a + b;
    printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }
```

De fiecare dată când programul apelează funcția, se alocă spațiu în stivă pentru a depune variabilele locale *a, b* și *c*. Când execuția funcției se încheie, se elimină valorile variabilelor locale și se anulează alocarea spațiului în stivă. Chiar dacă funcția declară multe variabile locale, valoarea fiecăreia este stocată în stivă.

DECLARAREA VARIABILELOR GLOBALE

C/C++ 234

În secțiunea 218, ați învățat că variabilele locale sunt variabile definite în cadrul unei funcții, numele și existența lor fiind recunoscute numai în acea funcție. Pe lângă variabilele locale, limbajul C permite programelor dumneavoastră să folosească variabile globale, ale căror nume, valori și existență sunt recunoscute în întregul program. Cu alte cuvinte, toate programele dumneavoastră în C pot folosi variabile globale. Următorul program, *global.c*, ilustrează folosirea a trei variabile globale, numite *a*, *b* și *c*:

```
#include <stdio.h>
int a = 1, b = 2, c = 3; // Variabile globale

void val_global(void)
  {
    printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }

void main(void)
  {
    val_global();
    printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }
```

Atunci când compilați și executați acest program, ambele funcții, *val_global* și *main*, vor afișa pe ecran valorile variabilelor globale. Observați că ați declarat variabilele în afara funcțiilor. Când declarați variabile globale în acest mod, toate funcțiile programului pot folosi și modifica valorile unei variabile globale prin simpla referire la numele său. Deși variabilele globale par convenabile la prima vedere, utilizarea lor necorespunzătoare poate conduce la erori foarte dificil de depanat, așa cum veți învăța în secțiunea 235.

EVITAREA UTILIZĂRII VARIABILELOR GLOBALE

C/C++ 235

În secțiunea 234, ați învățat să declarați variabile globale, pe care programul le recunoaște în toate funcțiile sale. La o primă privire, folosirea variabilelor globale pare a simplifica programarea, deoarece elimină necesitatea folosirii parametrilor în funcții și, mai important, necesitatea înțelegerii *apelului prin valoare* și a *apelului prin referință*. Din păcate, în loc să reducă numărul de erori, variabilele globale adesea îl sporesc. Deoarece codul dumneavoastră poate modifica valoarea unei variabile globale în oricare loc din program, este foarte dificil pentru alt programator care citește programul să găsească fiecare loc din program în care variabila globală se modifică. De aceea, alți programatori ar putea să modifice programul dumneavoastră fără să înțeleagă pe deplin efectul pe care modificarea îl are asupra variabilei globale. Ca regulă, funcțiile trebuie să modifice doar acele variabile care le sunt transmise ca parametri. În acest fel, programatorii pot să studieze prototipurile funcțiilor pentru a determina rapid care variabilă este modificată de o funcție.

Dacă programul dumneavoastră folosește variabile globale, ar fi bine să reconsiderați concepția programului. Scopul dumneavoastră trebuie să fie eliminarea (sau cel puțin minimizarea) folosirii variabilelor globale.

236 REZOLVAREA CONFLICTELOR DINTRE NUMELE VARIABILELOR GLOBALE ȘI CELE LOCALE



Așa cum ați învățat, variabilele locale sunt variabile pe care le declarați în cadrul unei funcții și al căror nume e recunoscut doar de acea funcție. Pe de altă parte, când declarați variabile globale, în afara funcțiilor, fiecare funcție din program va recunoaște numele lor. Dacă programul dumneavoastră folosește variabile globale, s-ar putea ca uneori numele unei variabile globale să fie același cu numele unei variabile locale, pe care programul dumneavoastră o declară în cadrul unei funcții. De exemplu, programul următor, *conflict.c*, folosește variabilele globale *a*, *b* și *c*. Funcția *conflict_a* folosește o variabilă locală numită *a* și variabilele globale *b* și *c*:

```
#include <stdio.h>
int a = 1, b = 2, c = 3; // Variable globale

void conflict_a(void)
  {
   int a = 100;

   printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }

void main(void)
  {
   conflict_a();
   printf("a contine %d b contine %d c contine %d\n", a, b, c);
  }
```

Când veți compila și executa programul *conflict.c*, pe ecran se vor afișa următoarele:

```
a contine 100 b contine 2 c contine 3
a contine 1 b contine 2 c contine 3
C:\>
```

Când numele variabilei globale intră în conflict cu cel al variabilei locale, compilatorul de C va folosi întotdeauna variabila locală. După cum puteți vedea, modificarea variabilei *a* de către funcția *conflict_a* are efect doar în cadrul funcției.

Observație: *Deși acest program are drept scop ilustrarea modului în care compilatorul de C rezolvă conflictele de nume, el scoate în evidență și confuzia care se poate produce când folosiți variabilele globale. În acest caz, un programator care citește codul dumneavoastră trebuie să fie foarte atent pentru a observa că funcția nu modifică variabila globală* ***a***, *ci pe cea locală.*

237 DEFINIREA ÎMBUNĂTĂȚITĂ A DOMENIULUI DE VALABILITATE A VARIABILELOR GLOBALE



În secțiunea 234, ați învățat că o variabilă globală este o variabilă pe care o recunosc toate funcțiile programului dumneavoastră. Alegând locul în care definiți o variabilă globală, puteți stabili funcțiile care se pot referi la respectiva variabilă. Cu alte cuvinte, puteți controla domeniul ei de valabilitate (*scope*). Când programul dumneavoastră declară o variabilă globală, orice funcție care urmează declarației poate face referință la această variabilă, până

la sfârșitul fișierului sursă. Funcțiile definite înaintea unei variabile globale nu o pot accesa. Să luăm de exemplu următorul program, *glob_dom*, care definește variabila globală *titlu*:

```
#include <stdio.h>

void titlu_necunosc(void)
  {
    printf("Titlul cartii e %s\n", titlu);
  }

char titlu[] = "Totul despre C/C++";

void main(void)
  {
    printf("Titlu: %s\n", titlu);
  }
```

După cum vedeți, funcția *titlu_necunosc* va încerca să afișeze variabila *titlu*. Dar pentru că declarația variabilei globale apare după definirea funcției, variabila globală nu este recunoscută în cadrul funcției. Când încercați să compilați acest program, compilatorul va genera o eroare. Pentru a o corecta, mutați declarația variabilei globale înaintea definirii funcției.

APELUL PRIN VALOARE

C/C++238

Așa cum ați învățat, programele dumneavoastră transmit informația funcțiilor folosind parametri. Când transmiteți un parametru unei funcții, compilatorul folosește o tehnică cunoscută sub numele de *apel prin valoare*, pentru a da funcției o copie a valorii parametrului. Folosind *apelul prin valoare*, orice modificare a parametrului va exista numai în interiorul funcției apelate. La sfârșitul execuției funcției, valoarea variabilelor care i-au fost transmise rămâne nemodificată în funcția apelantă. De exemplu, următorul program, *nemodif.c*, transmite trei parametri (variabilele *a*, *b* și *c*) funcției *afis_si_modif*, care va afișa valorile, le va adăuga 100 și apoi va afișa rezultatul. După execuția funcției, programul va afișa valorile variabilelor. Deoarece se folosește apelul prin valoare, funcția nu modifică valorile variabilelor în cadrul funcției apelante, cum se vede mai jos:

```
#include <stdio.h>

void afis_si_modif(int prima, int a_doua, int a_treia)
  {
    printf("Valorile originale ale functiei %d %d %d\n",
    prima, a_doua, a_treia);
    prima += 100;
    a_doua += 100;
    a_treia += 100;
    printf("Valorile finale ale functiei %d %d %d\n",
    prima, a_doua, a_treia);
  }

void main(void)
  {
    int a = 1, b = 2, c = 3;
```

```
    afis_si_modif(a, b, c);
    printf("Valorile finale in main %d %d %d\n", a, b, c);
}
```

Când compilați și executați programul *nemodif.c*, pe ecranul dumneavoastră vor fi afișate următoarele:

```
Valorile originale ale functiei 1 2 3
Valorile finale ale functiei 101 102 103
Valorile finale in main 1 2 3
C:\>
```

După cum puteți vedea, modificările variabilei sunt vizibile numai în cadrul funcției. După execuția ei, variabila dumneavoastră rămâne nemodificată în *main*.

Observație: *Când folosiți apelul prin referință (prezentat pe larg în secțiunea 240), o funcție poate să schimbe valoarea unui parametru astfel încât modificarea să fie vizibilă și în afara funcției.*

239 PREVENIREA MODIFICĂRII VALORII PARAMETRILOR CU APELUL PRIN VALOARE



În secțiunea 238, ați învățat că limbajul C folosește în mod prestabilit apelul prin valoare pentru a transmite parametrii funcțiilor. Ca rezultat, orice modificări ale valorilor parametrilor apar numai în interiorul funcției. La sfârșitul execuției funcției, valorile variabilelor transmise de program funcției rămân nemodificate. Așa cum s-a spus în secțiunea „Primele noțiuni de C", o variabilă este, în esență, un nume atribuit unei locații de memorie. Fiecare variabilă are două atribute importante: valoarea sa curentă și adresa de memorie. În cazul programului *nemodif.c*, prezentat în secțiunea 238, variabilele *a*, *b* și *c* ar putea utiliza adresele de memorie arătate în figura 239.1.

Adresa		Variabila
1000	1	a
1002	2	b
1004	3	c
	Memorie	

Figura 239.1 *Variabilele păstrează valorile și ocupă o anumită locație de memorie.*

Când transmiteți parametrii către o funcție, compilatorul de C plasează valorile corespunzătoare într-o stivă. În cazul variabilelor *a*, *b* și *c*, stiva conține valorile 1, 2 și 3. Când funcția accesează valorile variabilelor, funcția face referire la locațiile stivei, cum se arată în figura 239.2.

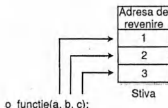


Figura 239.2 *Funcția face referință la valorile parametrilor stocate în stivă.*

Când funcția schimbă valorile unui parametru, modifică de fapt valorile din stivă, cum se vede în figura 239.3.

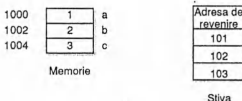


Figura 239.3 *Modificările pe care funcția le efectuează asupra valorilor parametrilor afectează numai valorile din stivă.*

Când se încheie execuția funcției, compilatorul descarcă valorile din stivă și nu mai sunt luate în considerare modificările pe care funcția le-a efectuat în locațiile stivei. Funcția nu face referire niciodată la locațiile de memorie care conțin valorile variabilelor, astfel că modificările parametrilor primiți în urma unui apel prin valoare nu vor mai exista după încheierea execuției funcției.

APELUL PRIN REFERINȚĂ

C/C++ 240

Așa cum ați învățat, compilatorul de C transmite parametrii funcțiilor folosind în mod prestabilit apelul prin valoare. Folosind apelul prin valoare, funcțiile nu pot modifica valoarea variabilei transmise. În cele mai multe programe, totuși, funcțiile dumneavoastră trebuie să modifice variabilele într-un mod sau altul. De exemplu, o funcție care citește date dintr-un fișier trebuie să le plaseze într-un tablou de șiruri de caractere. De asemenea, o funcție cum este *strupr* (prezentată în secțiunea „Șiruri") trebuie să convertească în majuscule literele dintr-un șir de caractere. În cazul funcțiilor care modifică valorile parametrilor, programul trebuie să îi transmită către funcție folosind *apelul prin referință*. Diferența între apelul prin valoare și apelul prin referință este că, folosind apelul prin valoare, funcția primește o copie a valorii unui parametru. Pe de altă parte, cu apelul prin referință, funcția primește adresa de memorie a variabilei. De aceea, funcțiile pot să modifice valoarea păstrată într-o anumită locație de memorie (cu alte cuvinte, valoarea variabilei), iar modificarea rămâne și după terminarea execuției funcției. Pentru a utiliza apelul prin referință, programul dumneavoastră trebuie să folosească *pointeri*. Aceștia sunt prezentați pe larg în secțiunea „Matrice, pointeri și structuri". Deocamdată, considerați un pointer, pur și simplu, ca o adresă de memorie. Pentru a atribui unui pointer adresa unei variabile, folosiți operatorul C de adresare (&). Pentru a accesa ulterior valoarea din locația de memorie pe

care o indică pointerul, folosiți operatorul C de redirectare (*). Secțiunile 241 și 242 explică în detaliu acești operatori.

241 OBȚINEREA UNEI ADRESE



O variabilă este, în esență, un nume atribuit uneia sau mai multor locații de memorie. Când programele dumneavoastră rulează, fiecare variabilă rămâne în locația sa proprie de memorie. Programul dumneavoastră localizează variabila în memorie folosind *adresa de memorie* a variabilei. Pentru a determina adresa unei variabile, folosiți operatorul de adresare (&). De exemplu, următorul program, *vezi_adr.c*, folosește operatorul de adresare pentru a fișa adresele variabilelor *a*, *b* și *c* (în format hexazecimal):

```
#include <stdio.h>

void main(void)
  {
    int a = 1, b = 2, c = 3;

    printf("Adresa lui a e %x valoarea lui a e %d\n", &a, a);
    printf("Adresa lui b e %x valoarea lui b e %d\n", &b, b);
    printf("Adresa lui c e %x valoarea lui c e %d\n", &c, c);
  }
```

Atunci când compilați și executați acest program, el va afișa pe ecran un rezultat similar cu următorul (valoarea reală a adresei poate să difere):

```
Adresa lui a e fff4 valoarea lui a e 1
Adresa lui b e fff2 valoarea lui a e 2
Adresa lui c e fff0 valoarea lui a e 3
C:\>
```

Când programul dumneavoastră va apela o funcție care trebuie să modifice valoarea unor variabile, va transmite variabilele prin referință (adresa de memorie), folosind operatorul de adresare, ca mai jos:

```
o_functie(&a, &b, &c);
```

242 UTILIZAREA ADRESEI VARIABILEI



În secțiunea 241, ați învățat cum se folosește operatorul C de adresare pentru a obține adresa de memorie a unei variabile. Când transmiteți o adresă unei funcții, trebuie să indicați compilatorului de C că acea funcție va utiliza pointerul (adresa de memorie), nu valoarea variabilei. Pentru a face aceasta, trebuie să declarați o *variabilă pointer*. Declarația unei variabile pointer este foarte asemănătoare cu cea a unei variabile obișnuite, în care specificați tipul și numele variabilei. Diferența, totuși, este că numele variabilei pointer este precedat de un asterisc (*). Următoarea declarație creează o variabilă pointer pentru valori de tip *int*, *float* și *char*:

```
int *i_pointer;
float *f_pointer;
char *c_pointer;
```

După ce declarați o variabilă pointer, trebuie să-i atribuiți o adresă de memorie. Următoarea instrucțiune, de exemplu, atribuie variabilei pointer *i_pointer* adresa variabilei de tip întreg *a*:

```
i_pointer = &a;
```

Apoi, pentru a folosi valoarea indicată de către variabila pointer, programul dumneavoastră trebuie să folosească operatorul C de redirectare – asteriscul (*). De exemplu, următoarea instrucțiune atribuie valoarea 5 variabilei *a* (a cărei adresă este conținută în variabila pointer *i_pointer*):

```
*i_pointer = 5;
```

În mod similar, următoarea instrucțiune atribuie variabilei *b* valoarea indicată de variabila *i_pointer*:

```
b = *i_pointer;
```

Când vreți să utilizați valoarea pe care o indică variabila pointer, folosiți operatorul de redirectare (*). Când doriți să atribuiți unei variabile pointer o adresă de variabilă, folosiți operatorul de adresare (&). Următorul program, *util_adr.c*, ilustrează modul de utilizare a variabilei pointer. Programul atribuie variabilei pointer *i_pointer* adresa variabilei *a*. Programul folosește apoi variabila pointer pentru a modifica, afișa și atribui valoarea variabilei:

```
#include <stdio.h>
void main(void)
  {
    int a = 1, b = 2;
    int *i_pointer;

    i_pointer = &a;  // Atribuie o adresa
    *i_pointer = 5;  // Modifica valoarea indicata de i_pointer
                     // in 5
    // Afiseaza valoarea
    printf("Valoarea indicata de i_pointer e %d
      variabila a e %d\n",*i_pointer, a);
    b = *i_pointer; // Atribuie valoarea
    printf("Valoarea lui b e %d\n", b);
    printf("Valoarea lui i_pointer %x\n", i_pointer);
  }
```

Amintiți-vă că pointerul nu este nimic mai mult decât o adresă de memorie. Programul dumneavoastră trebuie să atribuie valoarea indicată de pointer (adresă). Programul *util_adr.c* a atribuit pointerului adresa variabilei *a*, dar ar fi putut la fel de ușor să atribuie adresa variabilei *b*.

Observație: *Când folosiți pointeri, trebuie să țineți cont de tipul valorilor, cum ar fi* **int**, ***float*** *sau* **char**. *Programul dumneavoastră trebuie să atribuie numai adrese de valori întregi pointerilor de tip întreg și așa mai departe.*

243 MODIFICAREA VALORII PARAMETRILOR



Aşa cum aţi învăţat, pentru a modifica valoarea parametrilor în cadrul unei funcţii, programul dumneavoastră trebuie să folosească apelul prin referinţă, transmiţând adresa variabilei. În cadrul funcţiei, trebuie să folosiţi pointeri. Următorul program, *sch_para.c*, foloseşte pointeri şi adrese (apel prin referinţă) pentru a afişa şi apoi a modifica parametrii transmişi funcţiei *afis_modif*:

```
#include <stdio.h>
void afis_modif(int *prima, int *a_doua, int *a_treia)
  {
    printf("Valorile originale ale functiei %d %d %d\n",
    *prima, *a_doua, *a_treia);
    *prima += 100;
    *a_doua += 100;
    *a_treia += 100;
    printf("Valorile finale ale functiei %d %d %d\n",
    *prima, *a_doua, *a_treia);
  }
void main(void)
  {
    int a = 1, b = 2, c = 3;
    afis_modif (&a, &b, &c);
    printf("Valorile finale in main %d %d %d\n", a, b, c);
  }
```

După cum puteţi vedea, când programul apelează funcţia, îi transmite ca parametri adresele variabilelor *a*, *b* şi *c*. În cadrul programului *sch_para.c*, funcţia *afis_modif* foloseşte variabile pointer şi operatorul de redirectare pentru a modifica şi afişa valorile parametrilor. Atunci când compilaţi şi executaţi programul *sch_para.c*, pe ecran va apărea următorul rezultat:

```
Valorile originale ale functiei 1 2 3
Valorile finale ale functiei 101 102 103
Valorile finale in main 101 102 103
C:\>
```

244 MODIFICAREA VALORII UNOR ANUMIŢI PARAMETRI



Aşa cum aţi învăţat, funcţiile dumneavoastră pot modifica valoarea parametrilor folosind apelul prin referinţă. Secţiunea 243, de exemplu, prezintă funcţia *afis_modif*, care foloseşte apelul prin referinţă pentru a modifica valoarea fiecăruia dintre parametrii săi. În multe cazuri, funcţia dumneavoastră trebuie însă să modifice valoarea unui singur parametru, lăsând nemodificată valoarea celui de al doilea parametru. De exemplu, următorul program, *sch_prim.c*, foloseşte funcţia *modif_prim* pentru a atribui parametrului *prim* valoarea parametrului *al_doilea*:

```
#include <stdio.h>
void modif_prim(int *prim, int al_doilea)
  {
    *prim = al_doilea; // Atribuie primului valoarea celui
                       // de-al doilea
  }
void main(void)
  {
    int a = 0, b = 5;
    modif_prim(&a, b);
    printf("Valoarea lui a %d valoarea lui b %d\n", a, b);
  }
```

După cum puteți vedea, funcția *modif_prim* utilizează apelul prin referință pentru a modifica valoarea primului parametru și apelul prin valoare pentru al doilea parametru. Atunci când funcția folosește ambele tehnici – veți întâlni astfel de situații – trebuie să rețineți când se utilizează pointeri și când se face referire directă la variabilă. Ca regulă, parametrii a căror valoare vreți să o modificați necesită apelul prin referință. Pentru a înțelege mai bine efectul apelului prin referință față de cel prin valoare, să modificăm funcția *modif_prim*, ca mai jos:

```
void modif_prim(int *prim, int al_doilea)
  {
    *prim = al_doilea;  // Atribuie primului valoarea celui
                        // de al doilea
    al_doilea = 100;
  }
```

Atunci când compilați și executați acest program, veți vedea că valoarea primului parametru va fi modificată, dar al celui de al doilea nu. Deoarece al doilea parametru este transmis funcției cu apelul prin valoare, modificarea parametrului nu este vizibilă în afara funcției.

UTILIZAREA STIVEI LA APELUL PRIN REFERINȚĂ

C/C++ 245

Așa cum ați învățat, când compilatorul de C transmite parametrii către funcții, el plasează valorile lor într-o stivă. Compilatorul folosește stiva pentru a păstra parametrii fie că folosiți apelul prin valoare, fie că folosiți apelul prin referință. Când transmiteți un parametru prin valoare, compilatorul plasează *valoarea* parametrului în stivă. Când transmiteți parametrul prin referință, el va plasa *adresa* parametrului în stivă. Secțiunea 244 a prezentat programul *sch_prim.c*, care folosește funcția *modif_prim* pentru a atribui primului parametru valoarea celui de al doilea parametru al funcției. Atunci când programul apelează funcția, compilatorul plasează adresa variabilei *a* și valoarea variabilei *b* în stivă, cum se vede în figura 245.

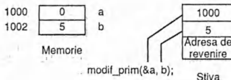


Figura 245 Compilatorul plasează în stivă o adresă și o valoare.

Deoarece funcția *modif_val* face, în realitate, referire la locația de memorie care conține valoarea variabilei *a*, modificarea variabilei va exista și în afara funcției.

246 PREZENTAREA VARIABILELOR STATICE



În limbajul C, variabilele pe care le declarați în cadrul funcției sunt adesea numite și *automatice*, deoarece compilatorul de C le creează automat când începe execuția funcției și apoi le distruge când ea se încheie. Această caracteristică a variabilelor se explică prin faptul că variabilele funcțiilor sunt păstrate de compilator temporar în stivă. Ca urmare, funcția atribuie o valoare unei variabile în timpul unei apelări, dar variabila pierde valorile după ce funcția se încheie. La următoarea apelare a funcției, valoarea variabilei este din nou nedefinită. În funcție de procesele efectuate de funcția dumneavoastră, e posibil ca variabilele funcțiilor să memoreze ultima valoare care le-a fost atribuită în cadrul funcției.

De exemplu, să presupunem că ați scris o funcție numită *print_medii*, care tipărește foaia matricolă pentru fiecare elev dintr-o școală. Să zicem că funcția folosește variabila *id_elev*, care păstrează numărul de identificare al elevului pentru care s-a tipărit ultima foaie matricolă. În acest fel, fără nici o mențiune, funcția va începe prin a tipări foaia matricolă a următorului elev. Pentru ca variabilele locale ale funcțiilor dumneavoastră să memoreze valorile în acest mod, trebuie să declarați variabilele utilizând cuvântul cheie *static*, ca mai jos:

```
void print_medii(int nr_print)
  {
    static int id_elev;
    // Alte instructiuni
  }
```

Următorul program, *static.c*, ilustrează utilizarea unei variabile *statice* în cadrul funcției. Programul care folosește funcția *print_medii* începe prin atribuirea valorii 100 variabilei *id_elev*. De fiecare dată când programul apelează funcția, ea va afișa valoarea variabilei și apoi o va mări cu 1, ca mai jos:

```
#include <stdio.h>

void print_medii(int nr_print)
  {
    static int id_elev = 100;
    printf("Se tiparesc mediile pentru elevul %d\n", id_elev);
      id_elev++;
    // Alte instructiuni
  }
```

```
void main(void)
  {
    print_medii(1);
    print_medii(1);
    print_medii(1);
  }
```

Atunci când compilați și executați programul *static.c*, pe ecran se vor afișa următoarele:

```
Se tiparesc mediile pentru elevul 100
Se tiparesc mediile pentru elevul 101
Se tiparesc mediile pentru elevul 102
C:\>
```

După cum puteți vedea, variabila *id_elev* își păstrează valoarea de la o apelare la următoarea.

Observație: *Când declarați variabile statice, compilatorul de C nu le stochează în stivă, ci în cadrul segmentului de date, astfel ca valorile lor să poată fi reținute.*

INIȚIALIZAREA VARIABILELOR STATICE

C/C++ 247

În secțiunea 246, ați învățat cum cuvântul cheie *static* cere compilatorului să rețină valorile variabilelor de la o apelare a unei funcții la următoarea. Atunci când declarați o variabilă statică în funcție, puteți să o inițializați, așa cum se vede mai jos:

```
void print_medii(int nr_printer)
  {
    static int id_elev = 100; // Initializata o singura data
    // Alte instructiuni
  }
```

Când declarați o variabilă ca *statică*, compilatorul de C va inițializa variabila cu valoarea pe care o indicați. Dacă apelați din nou funcția, mai târziu, compilatorul de C *nu va efectua din nou inițializarea*. Inițializarea unei variabile statice este diferită de celelalte inițializări de variabile pe care le execută de obicei compilatorul în cadrul funcțiilor. În cazul următoarei funcții, compilatorul va inițializa variabila *contor* de fiecare dată când programul apelează funcția:

```
void o_functie(int varsta, int *nume)
  {
    int contor = 1; // Initializata la fiecare apel

    // Alte instructiuni
  }
```

UTILIZAREA SECVENȚEI DE APELARE PASCAL

C/C++ 248

Pe măsură ce creați programe, puteți să utilizați funcții pe care le-ați creat anterior în limbajul Pascal. Ținând cont de tipul compilatorului, al editorului de legături și al bibliotecii cu care lucrați, puteți să apelați o funcție Pascal din programele dumneavoastră în C. Pașii pe care

trebuie să-i urmați pentru aceasta depind de compilatorul dumneavoastră. În plus, trebuie să introduceți la începutul programului dumneavoastră prototipul funcției, care să conțină cuvântul cheie *pascal*, cum se vede mai jos:

```
int pascal o_functie(int punctaj, int calificativ);
```

Dacă programul dumneavoastră rulează într-un mediu Windows, veți vedea că multe funcții de bibliotecă run-time folosesc secvența de apelare Pascal. Funcțiile care utilizează cuvântul cheie *pascal* nu acceptă un număr variabil de argumente (cum acceptă *printf* și *scanf*).

249 EFECTUL CUVÂNTULUI CHEIE PASCAL

C/C++

Ați învățat în secțiunea 231 că atunci când programul dumneavoastră apelează o funcție, compilatorul de C transmite parametrii către funcție folosind stiva. Compilatorul de C plasează parametrii în stivă de la dreapta la stânga. Figura 249.1 prezintă conținutul stivei în cazul unui apel de funcție C.

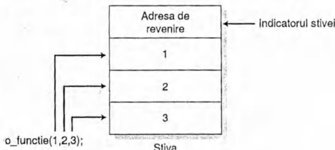


Figura 249.1 *Conținutul stivei în cazul unui apel de funcție C.*

Limbajul Pascal, pe de altă parte, pune argumentele în stivă de la stânga la dreapta. Figura 249.2 prezintă conținutul stivei cazul unui apel de funcție Pascal.

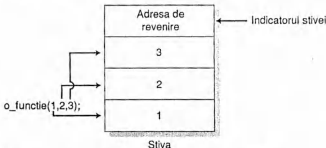


Figura 249.2 *Conținutul stivei în cazul unui apel de funcție Pascal.*

Dacă folosiți funcții Pascal în cadrul programelor dumneavoastră în C, folosiți cuvântul cheie *pascal* pentru a cere compilatorului de C să plaseze parametrii în stivă de la stânga la dreapta, în ordinea pe care o așteaptă limbajul Pascal.

SCRIEREA UNUI EXEMPLU DE LIMBAJ MIXT

C/C++250

Așa cum ați învățat, multe compilatoare de C vă permit să apelați funcții care au fost scrise în diferite alte limbaje de programare. Dacă apelați o funcție Pascal în cadrul programelor dumneavoastră în C, de exemplu, puteți introduce cuvântul cheie *pascal* în fața prototipului funcției. Așa cum ați văzut, acest cuvânt cheie cere compilatorului să depună parametrii în stivă de la stânga la dreapta. Pentru a analiza efectul lui, creați următoarea funcție, *vezi_val*, și introduceți înaintea ei cuvântul cheie *pascal*:

```
#include <stdio.h>
void pascal vezi_val(int a, int b, int c)
  {
    printf("a %d b %d c %d\n", a, b, c);
  }
```

Apoi, apelați funcția folosind următorul cod:

```
void main(main)
  {
    vezi_val(1, 2, 3);
    vezi_val(100, 200, 300);
  }
```

Ștergeți acum cuvântul cheie *pascal* și observați modificarea ordinii în care sunt afișate valorile parametrilor. Dacă programele dumneavoastră apelează o funcție Pascal, trebuie să utilizați cuvântul cheie *pascal* în prototipul funcției.

CUVÂNTUL CHEIE CDECL

C/C++251

În secțiunea 250, ați învățat că, dacă vreți să folosiți o funcție scrisă în limbajul Pascal, trebuie să folosiți cuvântul cheie *pascal*, astfel încât compilatorul să plaseze parametrii în stivă în ordinea corectă. Când folosiți funcții scrise în diferite limbaje de programare, puteți să includeți cuvântul cheie *cdecl* în cadrul prototipului funcțiilor dumneavoastră pentru a indica funcțiile în C și pentru a da mai multă claritate la citire. De exemplu, următorul prototip de funcție informează compilatorul că funcția *modif_val* folosește structura de apel a limbajului C:

```
int cdecl modif_val(int *, int *, int *);
```

Când compilatorul de C întâlnește cuvântul cheie *cdecl* în cadrul unei funcții, va avea grijă ca parametrii transmiși funcției să fie plasați în stivă de la dreapta la stânga. În plus, compilatorul se va asigura că editorul de legături folosește formatul C al numelui al funcției.

RECURSIVITATEA

C/C++252

Așa cum ați învățat, limbajul C vă permite să împărțiți programul dumneavoastră în părți mai mici, numite funcții. Folosind funcțiile, programele dumneavoastră vor deveni mai ușor de înțeles, de programat și de testat. În plus, puteți să folosiți funcții create pentru un program în cadrul altui program. În cursul execuției unui program, o funcție poate apela o altă

funcţie, care apelează la rândul ei alta, care, şi ea, poate apela multe alte funcţii. Într-o astfel de înlănţuire, fiecare funcţie execută o anumită operaţie. Dacă este cazul, limbajul C permite chiar ca o funcţie să se apeleze pe sine însăşi! O *funcţie recursivă* este o funcţie care se apelează pe sine însăşi pentru a efectua o anumită operaţie. Procesul în care o funcţie se apelează pe sine poartă numele de *recursivitate*. Pe măsură ce complexitatea programelor şi funcţiilor dumneavoastră va creşte, uneori va fi mai uşor să definiţi anumite operaţii prin referire la ele însele.

Când lucraţi cu programe şi funcţii complexe, pot apărea situaţii în care trebuie să creaţi o funcţie recursivă. Multe cărţi de programare dau ca exemplu problema factorialului pentru a ilustra modul în care funcţionează recursivitatea. Factorialul valorii 1 este 1. Factorialul valorii 2 este 2*1. Factorialul valorii 3 este 3*2*1. La fel, factorialul valorii 4 este 4*3*2*1. Procesul poate continua la infinit. Dacă privim cu atenţie, vom vedea că factorialul lui 4, de exemplu, este de patru ori factorialul lui 3 (3*2*1). La fel, factorialul lui 3 este de trei ori factorialul lui 2 (2*1). Factorialul lui 2 este de două ori factorialul lui 1(1). Tabelul 252 ilustrează procesul factorial.

Valoare	Calcul	Rezultat	Factorial
1	1	1	1
2	2*1	2	2*Factorial(1)
3	3*2*1	6	3*Factorial(2)
4	4*3*2*1	24	4*Factorial(3)
5	5*4*3*2*1	120	5*Factorial(4)

Tabelul 252 *Procesul factorial.*

Următorul program, *fact.c*, creează o funcţie recursivă, *factorial*, apoi foloseşte funcţia pentru a returna factorialul valorilor de la 1 la 5:

```
#include <stdio.h>

int factorial(int val)
  {
    if (val == 1)
      return(1);
    else
      return(val * factorial(val-1));
  }

void main(void)
  {
    int i;
    for (i = 1; i <= 5; i++)
      printf("Factorialul lui %d e %d\n", i, factorial(i));
  }
```

După cum puteţi vedea, funcţia *factorial* returnează un rezultat care se bazează pe propriul ei rezultat. Secţiunea 253 analizează funcţia *factorial* în detaliu.

FUNCȚIA RECURSIVĂ FACTORIAL

C/C++ 253

În secțiunea 252, ați învățat că o funcție recursivă este acea funcție care se apelează pe sine pentru a îndeplini o anumită sarcină. Secțiunea 252 v-a prezentat funcția *factorial* pentru a ilustra recursivitatea. Funcția *factorial* primește o anumită valoare a parametrului. Când începe execuția, mai întâi se verifică dacă valoarea este 1, al cărei factorial este prin definiție 1. Dacă valoarea este 1, funcția returnează valoarea 1. Dacă valoarea nu este 1, funcția returnează rezultatul înmulțirii dintre valoare și factorialul valorii minus 1.

Să presupunem, de exemplu, că programul apelează funcția cu valoarea 3. Funcția va returna rezultatul operației *3*factorial(3–1)*. Când compilatorul de C întâlnește apelul de funcție în cadrul instrucțiunii *return*, el apelează pentru a doua oară funcția, de această dată cu valoarea 3–1 (adică 2). Din nou, pentru că valoarea nu este 1, funcția returnează rezultatul operației *2*factorial(2–1)*. La a treia apelare, valoarea este 1. Ca urmare, funcția returnează valoarea 1 către funcția apelantă, care la rândul ei returnează rezultatul *2*1* către funcția care a apelat-o. Această funcție apelantă returnează apoi rezultatul operației *3*2*1* către funcția sa apelantă. Figura 253 prezintă acest lanț de apelări de funcții recursive și de returnării de valori în cazul apelului funcției *factorial(3)*.

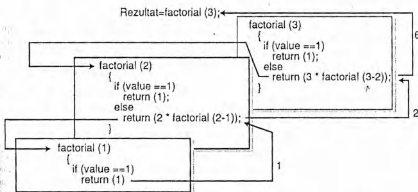


Figura 253 *Lanțul apelărilor de funcție și al întoarcerilor de valori pentru funcția recursivă **factorial**.*

O funcție recursivă este oarecum asemănătoare cu o structură ciclică, deoarece trebuie să precizați o condiție de încheiere. Dacă nu faceți această specificare, funcția nu se va sfârși niciodată. În problema factorialului, condiția de încheiere este factorial de 1 care, prin definiție, este 1.

SCRIEREA UNUI ALT EXEMPLU DE RECURSIVITATE

C/C++ 254

În secțiunea 252, ați învățat că o funcție recursivă este o funcție care se apelează pe sine cu scopul de a efectua un anumit proces. Secțiunea 253, la rândul său, a prezentat și explicat funcția recursivă *factorial*. Deoarece recursivitatea poate fi un concept dificil de înțeles, această secțiune prezintă încă un exemplu de funcție recursivă, *afis_invers*, care va afișa un șir de litere în ordine inversă. Dându-i literele ABCDE, funcția va afișa pe ecran literele EDCBA. Următorul program, *invers.c*, utilizează funcția *afis_invers*:

```
#include <stdio.h>

void afis_invers(char *sir)
  {
    if (*sir)
      {
        afis_invers(sir+1);
        putchar(*sir);
      }
  }

void main(void)
  {
    afis_invers("ABCDE");
  }
```

255 AFIȘAREA VALORILOR PENTRU A ÎNȚELEGE MAI BINE RECURSIVITATEA

C/C++

Așa cum ați învățat, o funcție *recursivă* este o funcție care se apelează pe sine pentru a efectua o anume operație. Secțiunea 252 a prezentat funcția recursivă *factorial*. Pentru a vă ajuta să înțelegeți mai bine procesul recursiv, programul *vezifact.c* include instrucțiunea *printf* în cadrul funcției *factorial*:

```
#include <stdio.h>

int factorial(int val)
  {
    printf("In functia factorial cu valoarea %d\n", val);
    if (val == 1)
      {
        printf("Returneaza valoarea 1\n");
        return(1);
      }
    else
      {
        printf("Returneaza %d * factorial(%d)\n", val, val-1);
        return(val * factorial(val-1));
      }
  }

void main(void)
  {
    printf("Factorialul lui 4 este %d\n", factorial(4));
  }
```

Atunci când compilați și executați programul *vezifact.c*, pe ecranul dumneavoastră vor fi afișate următoarele:

```
In functia factorial cu valoarea 4
Returneaza 4 * factorial(3)
In functia factorial cu valoarea 3
Returneaza 3 * factorial(2)
In functia factorial cu valoarea 2
Returneaza 2 * factorial(1)
In functia factorial cu valoarea 1
Returneaza valoarea 1
Factorialul lui 4 este 24
C:\>
```

Inserarea instrucțiunii *printf* la fiecare pas al funcției recursive vă ajută să înțelegeți mai bine procesul executat de funcție.

RECURSIVITATEA DIRECTĂ ȘI INDIRECTĂ

C/C++ 256

O funcție recursivă este o funcție care se apelează pe sine pentru a efectua o anumită operație. Mai multe secțiuni din acest capitol au prezentat funcții recursive. Când o funcție se apelează pe sine pentru a îndeplini o sarcină, funcția execută un *proces recursiv direct*. După ce veți examina câteva funcții recursive, veți fi în măsură să înțelegeți majoritatea funcțiilor care folosesc *recursivitatea directă*. Forma mai dificilă a recursivității, *recursivitatea indirectă*, are loc atunci când o funcție (funcția A) apelează o altă funcție (funcția B), care, la rândul său, apelează prima funcție (funcția A). Pentru că recursivitatea indirectă poate duce la realizarea unui cod dificil de înțeles, trebuie, ca regulă, să evitați utilizarea ei ori de câte ori este posibil.

DECIZIA DE UTILIZARE A RECURSIVITĂȚII

C/C++ 257

O funcție recursivă este o funcție care se apelează pe sine pentru a îndeplini o anumită sarcină. Atunci când creați funcții, puteți să utilizați recursivitatea pentru a rezolva elegant multe probleme. Totuși, din două motive, trebuie să evitați recursivitatea oriunde este posibil. În primul rând, funcțiile recursive pot fi dificil de înțeles pentru programatorul novice. În al doilea rând, de regulă, funcțiile recursive sunt adesea considerabil mai lente decât corespondentele lor nerecursive. Următorul program, *fara_rec.c*, apelează funcția nerecursivă *lung_sir* cu șirul de caractere „Totul despre C/C++" de 100000 de ori și afișează apoi perioada de timp cerută de acest proces:

```
#include <stdio.h>
#include <time.h>

int lung_sir(const char *sir)
  {
    int lung = 0;

    while (*sir++)
      lung++;
    return(lung);
  }

void main(void)
  {
```

```
   long int contor;
   time_t start_time, end_time;
   time(&start_time);
   for (contor = 0; contor < 100000L; contor++)
     lung_sir("Totul despre C/C++");
   time(&end_time);
   printf("Timpul de procesare %d\n", end_time - start_time);
 }
```

Următorul program, *ok_rec.c*, folosește o implementare recursivă a funcției *lung_sir* pentru a executa același proces:

```
#include <stdio.h>
#include <time.h>

int lung_sir(const char *sir)
 {
   if (*sir)
     return(1 + lung_sir(sir+1));
   else
     return(0);
 }

void main(void)
 {
   long int contor;
   time_t start_time, end_time;
   time(&start_time);
   for (contor = 0; contor < 100000L; contor++)
     lung_sir("Totul despre C/C++");
   time(&end_time);
   printf("Timpul de procesare %d\n", end_time - start_time);
 }
```

Înlocuiți numărul de apeluri ale funcției în cele două programe, de exemplu, cu 1 sau 2 milioane. Veți vedea că funcția nerecursivă se execută considerabil mai repede decât corespondenta ei recursivă. De aceea, când concepeți o funcție recursivă, rețineți că puteți adăuga o suprasarcină semnificativă pentru timpul de execuție al programului dumneavoastră.

258 DE CE SUNT LENTE FUNCȚIILE RECURSIVE



O funcție recursivă este o funcție care se apelează pe sine pentru a îndeplini o anumită sarcină. Așa cum ați învățat în secțiunea 257, un motiv pentru care e bine să evitați utilizarea recursivității este faptul că funcțiile recursive sunt, în general, mai lente decât omoloagele lor nerecursive. Funcțiile recursive sunt lente, pentru că la fiecare apel se introduce în program o *suprasarcină de apel*. Așa cum se arată în secțiunea 231, de fiecare dată când programul apelează funcția, compilatorul de C depune în stivă adresa instrucțiunii care urmează imediat după apelul funcției (numită *adresă de revenire*). Apoi, compilatorul depune în stivă

valorile parametrilor. Când execuția funcției se încheie, sistemul de operare al calculatorului descarcă adresele de revenire din stivă în contorul de program al CPU. Deși calculatorul poate să execute această operație de depunere și descărcare foarte repede, operațiile cer totuși timp.

Ca un exemplu, să presupunem că apelați funcția recursivă *factorial* cu valoarea 50. Funcția se va apela pe sine de 49 de ori. Dacă fiecare apel al funcției adaugă 10 milisecunde programului dumneavoastră, funcția va fi cu o jumătate de secundă mai lentă decât omologa sa nerecursivă, care nu încarcă decât cu o singură apelare a funcției. O suprasarcină de jumătate de secundă nu este mult, dar să presupunem că programul apelează funcția de zece ori. Jumătatea de secundă se va transforma rapid în cinci secunde. Dacă programul folosește funcția de 100 de ori, diferența de timp va fi de 50 de secunde, și așa mai departe. Dacă ați scris un program care cere un maxim de performanță, trebuie să încercați să eliminați funcțiile recursive oriunde este posibil.

Observație: *În cazul microprocesoarelor mai noi și mai rapide (cum sunt cele de 200 MHz), încetinirea sistemului de operare datorită funcțiilor recursive nu este atât de importantă cum era odată. Oricum, impactul funcțiilor recursive rămâne încă semnificativ și, oricând este posibil, trebuie să încercați să scrieți un cod eficace și lizibil fără să apelați la recursivitate.*

ELIMINAREA RECURSIVITĂȚII

C/C++ 259

O funcție recursivă este o funcție care se apelează pe sine pentru a îndeplini o anumită sarcină. Așa cum ați învățat, puteți mări performanța programului dumneavoastră folosind funcții nerecursive. Ca regulă, orice funcție care poate fi scrisă într-o formă recursivă poate fi scrisă, de asemenea, ca o structură ciclică, cum ar fi instrucțiunea *for* sau *while*. Următorul program, *loopfact.c*, folosește bucla *for* pentru a implementa funcția *factorial*:

```
#include <stdio.h>

int factorial(int val)
  {
    int rezultat = 1;
    int contor;

    for (contor = 2; contor <= val; contor++)
      rezultat *= contor;
    return(rezultat);
  }

void main(void)
  {
    int i;

    for (i = 1; i <= 5; i++)
      printf("Factorialul lui %d e %d\n", i, factorial(i));
  }
```

Când eliminați recursivitatea din cadrul programelor dumneavoastră folosind o structură ciclică, în general, ridicați nivelul de performanță. Rețineți însă că uitlizatorii vor înțelege mai ușor anumite operații din program când le implementați recursiv. La fel cum există situații în

care trebuie să găsiți calea de mijloc între viteza programului și consumul de memorie, există și unele în care trebuie să alegeți între lizibilitate și performanță.

260 TRANSMITEREA ȘIRURILOR CĂTRE FUNCȚII



Așa cum ați învățat, când transmiteți parametrii unei funcții, compilatorul de C, în mod prestabilit, îi transmite *prin valoare*. De aceea, modificarea lor nu se va păstra și în afara funcției. Pentru a modifica valorile parametrilor, trebuie să transmiteți parametrii *prin referință*. Excepția de la această regulă o face șirul de caractere. Atunci când apelați o funcție cu un șir de caractere, pur și simplu transmiteți funcției o matrice de octeți. Când compilatorul de C transmite o matrice (de orice tip, nu numai șir de caractere), el transmite funcției adresa de început a matricei. Cu alte cuvinte, compilatorul de C *folosește întotdeauna apelul prin referință pentru matrice*, astfel că nu trebuie să folosiți operatorul de *adresare*.

261 TRANSMITEREA ANUMITOR ELEMENTE DINTR-O MATRICE



Așa cum ați învățat în secțiunea 260, compilatorul de C transmite întotdeauna matricele către funcții folosind apelul prin referință. Atunci când lucrați cu șiruri de caractere, probabil că veți dori uneori ca o funcție să lucreze cu anumite elemente ale unei matrice. De exemplu, următorul program, *part_maj.c*, folosește funcția *strupr* pentru a converti o parte a unui șir de caractere în majuscule:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char alfabet[] = "abcdefghijklmnopqrstuvwxyz";
    strupr(&alfabet[13]);
    printf(alfabet);
  }
```

Funcția *strupr* așteaptă ca parametru adresa de început a unui șir de caractere terminat cu *NULL*. În acest caz, programul transmite funcției adresa literei n, căreia îi urmează câteva caractere terminate cu *NULL*. Prin transmiterea adresei unui element al matricei, programul dumneavoastră poate utiliza funcții pentru a manipula anumite elemente ale unei matrice.

262 CUVÂNTUL CHEIE CONST



Dacă analizați cu atenție prototipurile funcțiilor modificatoare de șiruri de caractere, prezentate în secțiunea „Șiruri", veți observa că multe dintre declarațiile parametrilor conțin cuvântul cheie *const* înaintea argumentelor de tip șir de caractere, ca mai jos:

```
char *strcpy(char *destinatie, char const *sursa);
```

În exemplul dat, definirea funcției *strcpy*, cuvântul cheie *const* precizează că funcția nu trebuie să modifice variabila *sursa* în cadrul funcției. Atunci când codul funcției încearcă să

schimbe valoarea șirului de caractere, compilatorul va genera o eroare. Următorul program, *sir_cons.c*, folosește cuvântul cheie *const* pentru parametrul *sir*:

```
#include <stdio.h>

void nu_modif(const char *sir)
  {
    while (*sir)
      *sir++ = toupper(*sir);
  }

void main(void)
  {
    char titlu[] = "Totul despre C/C++";

    nu_modif(titlu);
    printf(titlu);
  }
```

După cum puteți vedea, funcția *nu_modif* încearcă să convertească literele șirului de caractere în majuscule. Însă, pentru că programul folosește cuvântul cheie *const*, compilatorul va afișa un mesaj de eroare și codul nu va reuși compilarea. Dacă vreți să nu se modifice valoarea unui parametru pe care funcția îl primește prin referință, trebuie să folosiți cuvântul cheie *const* înaintea parametrului. Deoarece compilatorul de C, în mod prestabilit, transmite prin valoare parametrii care nu sunt pointeri, de obicei nu este nevoie de cuvântul cheie *const* în fața acestora.

PREVENIREA MODIFICĂRII PARAMETRILOR C/C++ 263

Așa cum ați învățat în secțiunea 262, cuvântul cheie *const* informează compilatorul că funcția nu trebuie să modifice valoarea unui anumit parametru. Dacă funcția încearcă să modifice valoarea parametrului, compilatorul va genera o eroare și programul nu se va compila. Trebuie să rețineți că deși în definirea funcției se precizează că un parametru este o constantă, nu înseamnă că funcția nu poate să-i modifice totuși valoarea. Următorul program, *schconst.c*, folosește un pointer la parametrul constant *sir* pentru a converti conținutul șirului în majuscule:

```
#include <stdio.h>
#include <ctype.h>

void nu_modif(const char *sir)
  {
    char *alias = sir;

    while (*alias)
      *alias++ = toupper(*alias);
  }

void main(void)
  {
    char titlu[] = "Totul despre C/C++";
    nu_modif(titlu);
```

```
    printf(titlu);
  }
```

Atunci când compilați și executați programul *schconst.c*, funcția *nu_modif* va converti caracterele șirului în majuscule. Deoarece ați folosit un *alias al pointerului* (o referire la locația de memorie a variabilei folosind alt nume), compilatorul nu va detecta modificarea valorii parametrului. În funcție de tipul compilatorului dumneavoastră, acesta poate genera un mesaj de avertizare. Atunci când creați propriile dumneavoastră funcții, nu folosiți aliasuri pentru a modifica valoarea parametrilor, așa cum se face în programul *schconst.c*. Dacă parametrul este cu adevărat constant, valoarea sa nu trebuie să se modifice. Programul din cadrul acestei secțiuni ar trebui să vă învețe că, în realitate, cuvântul cheie *const* nu poate preveni modificarea valorii parametrului.

264 DECLARAREA ȘIRURILOR NELIMITATE



În limbajul C, șirul de caractere este un tablou cu valori de tip *char*. Ați învățat în secțiunea „Șiruri" că, pentru a crea un șir de caractere, precizați numărul maxim de caractere pe care șirul le va avea vreodată, ca mai jos:

```
char nume[64];
char titlu[32];
char buffer[512];
```

Când transmiteți un șir de caractere către o funcție, în realitate transmiteți adresa de început a șirului. Pentru că șirul de caractere se termină cu caracterul *NULL*, funcțiile limbajului C nu acordă atenție numărului de caractere conținute într-un șir. Ca urmare, multe funcții declară parametrii de tip șir de caractere ca pe un tablou nelimitat (tablou a cărui dimensiune nu se precizează), ca mai jos:

```
int strlen(char sir[])
```

Declarația *char sir[]* spune compilatorului că funcția va primi pointerul unui șir de caractere terminat în *NULL*. Șirul poate avea 64 de caractere, 1024 de caractere sau, poate, numai caracterul *NULL*. Următorul program, *strtabl.c*, folosește un tablou nelimitat pentru a implementa funcția *strlen*:

```
#include <stdio.h>

int strlen(char sir[])
  {
    int i = 0;
    while (sir[i] != NULL)
      i++;
    return(i);
  }

void main(void)
  {
    printf("Lungimea sirului ABC e %d\n", strlen("ABC"));
    printf("Lungimea sirului Totul despre C/C++
```

```
    e %d\n",strlen("Totul despre C/C++"));
    printf("Lungimea sirului NULL e %d\n", strlen(""));
}
```

Când compilați și executați programul *strtabl.c*, veți vedea că funcția lucrează pentru orice dimensiune a șirului. Însă, ca majoritatea funcțiilor care lucrează cu șiruri de caractere, funcția va eșua în cazul unui șir care nu se termină cu caracterul *NULL*.

UTILIZAREA POINTERILOR FAȚĂ DE DECLARAREA ȘIRURILOR DE CARACTERE

C/C++ 265

Dacă studiați diferite funcții C care gestionează șiruri de caractere, puteți observa că șirurile sunt declarate fie ca tablouri nelimitate, fie ca pointeri, așa cum se vede mai jos:

```
char *strcpy(char destinatie[], char sursa[]);
char *strcpy(char *destinatie, char *sursa);
```

Ambele declarații din exemplul precedent informează compilatorul că lucrează cu șiruri de caractere. Ambele sunt identice din punct de vedere al funcționalității și ambele sunt corecte. Dacă vă creați propriile dumneavoastră funcții, formatul pe care îl alegeți trebuie să depindă de modul în care faceți referirea la parametri în cadrul funcției. Dacă tratați parametrii ca pointeri, utilizați modalitatea de declarare cu pointeri. Dacă însă tratați parametrii ca tablouri, folosiți tabloul. Tratând parametrii consecvent într-o anume modalitate, veți face ca programele dumneavoastră să fie mai ușor de înțeles.

FOLOSIREA STIVEI PENTRU PARAMETRII DE TIP ȘIR

C/C++ 266

Așa cum ați învățat, când programul transmite un parametru către o funcție, compilatorul de C plasează în stivă valoarea sau adresa parametrului. Când transmiteți unei funcții un șir de caractere, compilatorul de C plasează în stivă adresa de început a șirului. De exemplu, secțiunea 264 a prezentat programul *strtabl.c*, care transmite mai multe șiruri funcției *strlen*. Figura 266 prezintă datele pe care compilatorul le plasează în stivă la prima apelare a funcției.

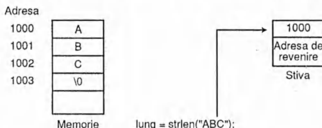


Figura 266 *Modalitatea de transmitere a parametrilor de tip șir de caractere către funcții.*

Așa cum puteți vedea, compilatorul nu plasează caracterele șirului în stivă, ci pur și simplu plasează adresa șirului terminat în *NULL*. Deoarece funcția primește numai o adresă (nu un tablou de octeți), nu contează câte caractere conține efectiv șirul.

267 VARIABILELE EXTERNE

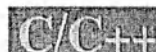


Adesea, veți utiliza într-un program funcții create în alt program. Pentru a simplifica reutilizarea funcțiilor, programatorii le plasează deseori în *biblioteci de coduri obiect*. Capitolul „Instrumente" discută despre astfel de biblioteci. În anumite cazuri, o bibliotecă poate defini o variabilă globală, cum sunt *_fmode*, *_psp* sau *errno*, variabile discutate în cuprinsul cărții. Când un cod din afara programului curent definește o variabilă globală pe care doriți să o utilizați în programul dumneavoastră, trebuie să declarați variabila folosint cuvântul cheie *extern*. Cuvântul cheie *extern* anunță compilatorul că variabila respectivă a fost declarată extern (în afara fișierului sursă curent) de un alt program. De exemplu, dacă examinați fișierul antet *dos.h*, veți întâlni numeroase declarații de variabile externe, cum sunt cele de mai jos:

```
extern int const _Cdecl _8087;
extern int _Cdecl _argc;
extern char **_Cdecl _argv;
extern char **_Cdecl environ;
```

Dacă nu utilizați cuvântul cheie *extern*, compilatorul va presupune că ați creat o variabilă cu numele specificat. Pe de altă parte, dacă includeți cuvântul cheie *extern*, compilatorul va căuta variabila globală specificată.

268 UTILIZAREA VARIABILELOR EXTERNE



Secțiunea 267 v-a prezentat cuvântul cheie *extern*, pe care îl veți utiliza în program pentru a anunța compilatorul că trebuie să folosescă o variabilă globală declarată într-un alt program, în afara programului curent. Pentru a înțelege mai bine cum funcționează cuvântul cheie *extern*, compilați fișierul *extern.c*, care conține declarația variabilei *nr_cap* și funcția *vezi_titlu*:

```
#include <stdio.h>

int nr_cap = 1500; // Variabila globala

void vezi_titlu(void)
  {
    printf("Totul despre C/C++");
  }
```

Atunci când compilați programul *extern.c*, compilatorul va crea fișierul *extern.obj*. Programul *veziext.c*, prezentat mai jos, utilizează variabila externă *nr_cap* din fișierul *extern.obj*:

```
#include <stdio.h>

void main(void)
  {
    extern int nr_cap;
    printf("Numarul de capitole e %d\n", nr_cap);
  }
```

Atunci când compilați programul *veziext.c*, executați următorii pași (dacă nu folosiți compilatorul *Turbo C++ Lite*, citiți documentația compilatorului respectiv):

1. Selectați din meniul Project opțiunea Open Project.
2. Alegeți directorul care conține programul *veziext.c* și introduceți numele de proiect *veziext*. Executați clic cu mouse-ul pe OK pentru a crea proiectul.
3. Selectați din meniul Project opțiunea Add Item.
4. Adăugați fișierul *extern.obj* la proiect.
5. Adăugați fișierul *veziext.c* la proiect.
6. Selectați din meniul Compile opțiunea Build All pentru a construi fișierul.

În acest caz, programul *veziext.c* afișează valoarea variabilei externe *nr_cap*. Programul nu utilizează funcția *vezi_titlu*, deși ar fi putut să o folosească prin simpla ei apelare. Scopul programului a fost însă ilustrarea utilizării cuvântului cheie *extern*.

Observație: *Pentru a utiliza variabile externe în alte compilatoare, consultați documentația on-line a compilatorului sau documentația scrisă livrată împreună cu el.*

VARIABILELE STATICE EXTERNE

C/C++ 269

În secțiunea 267, ați văzut cum cuvântul cheie *extern* anunță compilatorul că trebuie să facă referire la o variabilă globală definită într-un alt fișier decât programul curent. Când editorul de legături leagă modulele programului dumneavoastră, el va determina locația variabilei externe în memorie. În secțiunea 268, ați utilizat variabila globală *nr_cap*, definită în fișierul obiect *extern.obj*. Programul *veziext.c* poate accesa variabila, deoarece se referă la ea utilizând cuvântul cheie *extern*. Uneori pot apărea situații în care utilizați variabile globale și nu doriți ca acestea să fie accesate de funcțiile din afara fișierului obiect. În aceste cazuri, nu trebuie decât să introduceți cuvântul cheie *static* în fața numelui variabilei:

```
static int nume_var;
```

Fișierul următor, *stext.c*, declară două variabile globale, prima cu numele *nr_cap* și cealaltă cu numele *titlu*:

```
#include <stdio.h>
int nr_cap = 1500; // Variabila globala
static char titlu[] = "Totul despre C/C++";
void vezi_titlu(void)
  {
    printf(titlu);
  }
```

Compilați fișierul *stext.c*, pentru a crea fișierul obiect *stext.obj*. Apoi, creați următorul program, *nestatic.c*, care încearcă să utilizeze ambele variabile globale conținute în cadrul fișierului *stext.obj*:

```
#include <stdio.h>

void main(void)
  {
    extern int nr_cap;
    extern char *titlu;
    void vezi_titlu(void);

    printf("Numarul de capitole e %d\n", nr_cap);
    printf("Titlul cartii este %s\n", titlu);
    vezi_titlu();
  }
```

Așa cum ați învățat în secțiunea 268, pentru compilarea și legarea programului cu *Turbo C++ Lite*, executați următorii pași:

1. Selectați din meniul Project opțiunea Open Project.
2. Alegeți directorul care conține programul *nestatic.c* și introduceți numele de proiect *nestatic*. Executați clic cu mouse-ul pe OK pentru a crea proiectul.
3. Selectați din meniul Project opțiunea Add Item.
4. Adăugați fișierul *stext.obj* la proiect.
5. Adăugați fișierul *nestatic.c* la proiect.
6. Selectați din meniul Compile opțiunea Build All pentru a construi fișierul.

Atunci când compilați și legați programul *nestatic.c*, editorul de legături trebuie să afișeze un mesaj care anunță că compilatorul nu poate rezolva variabila *titlu*. Deoarece declarația variabilei *titlu* este precedată de cuvântul cheie *static*, variabila nu este cunoscută decât în cadrul fișierului obiect *stext.obj*.

270 CUVÂNTUL CHEIE VOLATILE



Pe măsura creșterii complexității programelor dumneavoastră, veți dori eventual să scrieți funcții și rutine de nivel inferior, care accesează porturile de intrare-ieșire sau care deservesc registrele de întrerupere ale PC-ului (denumite uneori pur și simplu *întreruperi*). Când programul dumneavoastră execută astfel de operații, utilizarea unei întreruperi sau accesarea unui port poate modifica variabilele care corespund unor locații de memorie sau adrese de port specifice. Deoarece aceste variabile pot fi modificate atât de programul dumneavoastră, cât și de numeroși factori externi programului, trebuie să anunțați compilatorul că valoarea variabilei se poate schimba oricând. Pentru a informa compilatorul că valoarea variabilei poate fi modificată în cadrul unor operații din afara programului, folosiți cuvântul cheie *volatile*:

```
volatile int variabila;
```

Când compilatorul întâlnește cuvântul cheie *volatile*, știe că nu poate să facă presupuneri asupra valorii variabilei la un anumit moment. De exemplu, compilatorul nu va plasa valoarea variabilei într-un registru pentru acces rapid. Procedând astfel, există riscul ca valoarea din registru să nu fie similară conținutului variabilei din memorie, pe care o

întrerupere, de exemplu, poate să o modifice fără știința programului după stocarea variabilei în registru. De aceea, când programul dumneavoastră trebuie să acceseze valoarea unei variabile, compilatorul va face referire la locația de memorie a variabilei.

***Observație:** În general, este bine să declarați variabilele **volatile** ca variabile globale. În acest mod, programele și operațiile exterioare fac referire la locațiile de memorie conținute în segmentul de date al programului, și nu la locațiile din stivă, pe care programul le descarcă atunci când funcția corespunzătoare se încheie.*

CADRUL DE APELARE ȘI INDICATORUL DE BAZĂ

C/C++ 271

Ați învățat că atunci când programul dumneavoastră apelează o funcție, compilatorul de C depune adresa de revenire și parametrii funcției în stivă. Informația din stivă referitoare la apelul de funcție este referită de C ca un *cadru de apelare*. Pentru ca funcțiile să găsească mai rapid cadrul de apelare, compilatorul de C atribuie registrul indicatorului de bază (BP) la adresa de început a cadrului. De asemenea, compilatorul de C plasează în stivă (în interiorul cadrului de apelare) și variabilele locale ale funcției. Figura 271 prezintă conținutul unui cadru de apelare simplu.

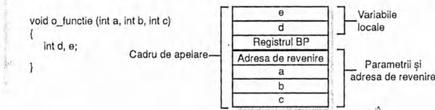


***Figura 271** Informația pe care compilatorul de C o plasează în stivă pentru apelarea unei funcții constituie cadrul de apelare.*

Dacă scrieți funcții în limbaj de asamblare pentru a le apela din interiorul programelor în C, trebuie să înțelegeți utilizarea și structura unui cadru de apelare, în așa fel încât funcțiile în limbaj de asamblare să poată accesa valorile parametrilor stocate în cadrul de apelare.

APELAREA UNEI FUNCȚII ÎN LIMBAJ DE ASAMBLARE

C/C++ 272

În secțiunea 236, ați învățat că programele pot apela funcții scrise în alte limbaje de programare, cum ar fi Pascal. În plus, programele dumneavoastră pot apela rutine scrise în limbaj de asamblare. Următoarea rutină în limbaj de asamblare, *swap_values*, schimbă între ele valorile a două variabile transmise funcției prin referință (prin adresă):

```
    .MODEL          small
    .CODE
    PUBLIC          _swap_values
_swap_values      PROC
    push            bp
    mov             bp, sp
```

```
        sub             sp,2
        push            si
        push            di

        mov             si,word ptr [bp+4]      ;Arg1
        mov             di,word ptr[bp+6]       ;Arg2

        mov             ax,word ptr [si]
        mov             word ptr [bp-2],ax

        mov             ax, word ptr [di]
        mov             word ptr [si],ax

        mov             ax,word ptr [bp-2]
        mov             word ptr [di],ax

        pop             di
        pop             si
        mov             sp,bp
        pop             bp
quit:                   ret

_swap_values            ENDP
        END
```

Compact-discul care însoțește cartea de față conține fișierul *swap.asm*. Dacă utilizați Borland C++, asamblați fișierul pentru a crea fișierul obiect *swap.obj*, ca mai jos:

```
C:\>TASM SWAP.ASM <ENTER>
```

Apoi, creați următorul program în C, *swap.c*, care folosește funcția *swap_val*:

```
#include <stdio.h>

void swap_val(int *, int *);

void main(void)
  {
    int a = 1, b = 2;

    printf("Valorile originale a %d b %d\n", a, b);
    swap_val(&a, &b);
    printf("Valorile dupa operatia de schimb a %d b %d\n", a, b);
  }
```

În acest caz, ați scris funcția *swap_val* cu pointeri *near*. Dacă schimbați modelul de memorie, trebuie să modificați și rutina în limbaj de asamblare.

273 RETURNAREA UNEI VALORI DINTR-O FUNCȚIE ÎN LIMBAJ DE ASAMBLARE



În secțiunea 271, ați învățat cum să apelați o funcție în limbaj de asamblare dintr-un program în C. În programul *swap.asm*, funcția nu returnează nici un rezultat. Următoarea rutină în

limbaj de asamblare, *_get_maximum*, returnează însă valoarea cea mai mare dintre două numere întregi:

```
        .MODEL small
        .CODE
        PUBLIC _get_maximum
_get_maximum    PROC
        push    bp
        mov     bp, sp
Arg1            equ         [bp+4]
Arg2            equ         [bp+6]
        mov     ax,Arg1             ; Muta Arg1 in AX
        mov     Arg2,ax             ; Compara Arg2 cu Arg1
        jp      arg2_maimare        ; Sare daca Arg2 este mai mare
        jmp     final
arg2_maimare:   mov         ax,Arg2
final: pop      bp
                ret
_get_maximum    ENDP
        END
```

Compact-discul care însoțește această carte conține fișierul *get_max.asm*, care include rutina *get_maximum*. După cum puteți vedea, rutina în limbaj de asamblare își plasează rezultatul în registrul AX. Capitolele următoare vor explica în detaliu diferitele registre; deocamdată, puteți considera registrul AX similar cu registrul BP, prezentat într-o secțiune anterioară. Următorul program, *reda_max.c*, apelează funcția în limbaj de asamblare pentru a determina maximul dintre două valori:

```
#include <stdio.h>
extern int _get_maximum(int, int);
void main(void)
  {
    int rezultat;
    rezultat = _get_maximum(100, 200);
    printf("Cea mai mare valoare e %d\n", rezultat);
  }
```

Atunci când programul apelează funcția, compilatorul atribuie valoarea registrului AX ca rezultat al funcției.

FUNCȚII CARE NU RETURNEAZĂ VALORI

C/C++ 274

Pe măsură ce veți crea diverse funcții, probabil că veți scrie și una care nu returnează nici o valoare. Așa cum ați învățat, compilatorul de C, dacă nu i se precizează altfel, presupune că o funcție returnează o valoare de tip *int*. Dacă funcția dumneavoastră nu returnează nici o valoare, trebuie să o declarați de tip *void*, ca mai jos:

```
void fctia_mea(int varsta, char *nume);
```

Dacă programul încearcă mai târziu să folosească valoarea returnată de funcție, ca mai jos, compilatorul va genera eroare:

```
rezultat = fctia_mea(32, "Jamsa");
```

275 FUNCȚIILE CARE NU UTILIZEAZĂ PARAMETRI C/C++

Pe măsură ce veți crea diverse funcții și programe, probabil că veți scrie și una care nu utilizează nici un parametru. Atunci când definiți funcția (și prototipul funcției), trebuie să utilizați cuvântul cheie *void* pentru a informa compilatorul (și pe alți programatori) că funcția dumneavoastră nu utilizează parametri:

```
int fctia_mea(void);
```

Dacă programul va încerca mai târziu să apeleze funcția cu parametri, compilatorul va genera o eroare.

276 CUVÂNTUL CHEIE AUTO C/C++

Dacă veți examina diferite programe în C, probabil că veți întâlni declarații de variabile care utilizează cuvântul cheie *auto*, ca mai jos:

```
auto int contor;
auto int flag;
```

Cuvântul cheie *auto* informează compilatorul că variabila funcției este locală și că el trebuie să o creeze și să o distrugă automat. Compilatorul creează variabile automatice alocând spațiu în stivă. Deoarece variabilele sunt în mod prestabilit automatice, cuvântul cheie *auto* nu apare de obicei în programe. În cadrul unei funcții, următoarele declarații de variabile sunt identice:

```
auto int contor;
int contor;
```

277 DOMENIUL DE VALABILITATE C/C++

În cadrul programelor dumneavoastră, funcțiile și variabilele au un *domeniu de valabilitate* (scope), care definește aria din program în care numele lor au semnificație. De exemplu, să considerăm următorul program, *douacont.c*, care utilizează două variabile denumite *contor*.

```
#include <stdio.h>

void beep(int nr_beep)
  {
    int contor;
    for (contor = 1; contor <= nr_beep; contor++)
```

```
    putchar(7);
  }
void main(void)
  {
    int contor;
    for (contor = 1; contor <= 3; contor++)
      {
        printf("Gata sa emita un sunet de %d ori\n", contor);
        beep(contor);
      }
  }
```

După cum puteți vedea, ambele funcții, *beep* și *main*, utilizează variabile denumite *contor*. Pentru compilatorul de C, cele două variabile sunt totuși distincte – fiecare are un domeniu de valabilitate diferit. În cazul funcției *beep*, se recunoaște variabila sa *contor* (cu alte cuvinte, *contor* are un domeniu bine definit) numai atâta timp cât se execută funcția. Asemănător, în cazul funcției *main*, variabila sa *contor*, are înțeles numai cât timp se execută funcția principală. Ca rezultat, bucla *for* care modifică variabila *contor* în funcția *beep* nu are nici un efect asupra variabilei *contor* din funcția *main*.

Atunci când se vorbește despre domeniul unei variabile, se utilizează frecvent termenii *variabile locale* și *variabile globale*. O variabilă locală este o variabilă al cărei domeniu este restricționat la o anumită funcție. Pe de altă parte, o variabilă globală este recunoscută în tot programul. În cazul programului *douacont.c*, fiecare funcție definește apariția variabilei *contor* ca locală.

CATEGORIILE DE DOMENII ÎN C

C/C++ 278

Așa cum ați învățat, *domeniul de valabilitate* al unui identificator (de obicei un nume de funcție sau variabilă) este partea din program în cadrul căreia identificatorul are înțeles (cu alte cuvinte, partea unde programul poate utiliza identificatorul). Limbajul C cunoaște patru categorii de domenii: al unui bloc, al unei funcții, al unui prototip de funcție și al unui fișier. În plus, C++ definește domeniul unei clase. Domeniul de tip *bloc* reprezintă regiunea încadrată de acolade în care programul dumneavoastră a definit o variabilă. De obicei, domeniul de tip *bloc* se referă la o funcție. Variabilele locale au, de obicei, domenii de tip bloc. Așa cum ați învățat în capitolul „Primele noțiuni de C", puteți să declarați variabile după orice acoladă deschisă. Domeniul unei variabile va continua până la acolada de închidere, aceasta însemnând că un parametru poate avea ca domeniu numai, de exemplu, interiorul unei structuri *if*. Parametrii formali au domeniul de tip bloc limitat la funcția care definește respectivul parametru. Domeniul de tip *funcție* definește regiunea dintre acolada de început și cea de sfârșit a unei funcții. Unicul articol cu domeniu de tip funcție este o etichetă utilizată de instrucțiunea *goto*. Domeniul de tip *prototip de funcție* specifică regiunea dintre începutul și sfârșitul unui prototip de funcție. Identificatorii care apar într-un prototip de funcție au un înțeles numai în cadrul acelui prototip, ca mai jos:

```
int oarecare(int varsta, char *nume);
```

Domeniul de tip *fișier* specifică regiunea dintre declarația unui identificator până la sfârșitul fișierului sursă. Variabilele globale au domeniu de tip fișier, ceea ce înseamnă că numai funcțiile care sunt scrise după declarația unei variabile globale se pot referi la acea variabilă globală. În C++, domeniul de tip *clasă* definește colecția de metode și de structuri de date care alcătuiesc clasa respectivă.

279 SPAȚIUL DE NUME ȘI IDENTIFICATORII



Așa cum ați învățat, *domeniul de valabilitate* definește regiunea din program în care un identificator are semnificație. În mod similar, *spațiul de nume* definește regiunea în care numele de identificator trebuie să fie unice. În sensul cel mai simplu, un *identificator* este un nume. Limbajul C definește patru clase de identificatori, prezentate de lista următoare:

- Etichetele *goto*: numele de etichetă care urmează unei instrucțiuni *goto* trebuie să fie unic în cadrul unei funcții.
- Etichetele de structură, uniune sau enumerare: numele tipurilor structură, uniune sau enumerare trebuie să fie unice în cadrul unui bloc.
- Numele de membru al unei structuri sau uniuni: numele de membru care apar în cadrul unei structuri sau uniuni trebuie să fie unice. Structurile sau uniunile diferite pot avea nume de membru identice.
- Variabilele, identificatorii *typedef*, funcțiile și membrii enumerați: acești identificatori trebuie să fie unici în cadrul domeniului de vizibilitate în care sunt definiți (vezi secțiunea 278).

280 VIZIBILITATEA UNUI IDENTIFICATOR



Așa cum ați învățat, domeniul de valabilitate (scope) definește zona din program în care un identificator are înțeles. În mod similar, *vizibilitatea* unui identificator definește porțiunea de cod în care un program poate accesa identificatorul. De regulă, domeniul de valabilitate al unui identificator și vizibilitatea sa sunt aceleași, însă, când programul dumneavoastră declară un identificator cu același nume într-un bloc aflat în interiorul domeniului de valabilitate al unui identificator, compilatorul de C ascunde temporar identificatorul exterior (cu alte cuvinte, identificatorul exterior își pierde vizibilitatea, iar compilatorul nu îl recunoaște). Să considerăm următorul program, *vizibil.c*, care folosește doi identificatori numiți *valoare*:

```
#include <stdio.h>
void main(void)
  {
    int valoare = 1500;
    if (valoare > 1499)
      {
        int valoare = 1;
        printf("Valoarea interioara e %d\n", valoare);
      }
    printf("Valoarea exterioara e %d\n", valoare);
  }
```

Atunci când compilați și executați programul *vizibil.c*, pe ecran se va afișa următoarea ieșire:

```
Valoarea interioara e 1
Valoarea exterioara e 1500
C:\>
```

Când programul declară variabila *valoare* în interiorul instrucțiunii *if*, declarația variabilei cere compilatorului să ascundă apariția exterioară a variabilei cu același nume. În afara blocului, variabila exterioară devine însă vizibilă din nou compilatorului.

DURATA

C/C++ 281

Când se discută despre variabile, *durata* semnifică perioada de timp în care unui identificator i se alocă un spațiu în memoria sistemului. Limbajul C suportă trei tipuri de durată: *locală*, *statică* și *dinamică*. Variabilele automatice, create în cursul apelării funcției, sau variabilele definite în cadrul unui bloc de instrucțiuni au *durată locală*. Programele dumneavoastră trebuie întotdeauna să inițializeze variabilele locale. Dacă nu inițializează o variabilă locală, atunci programul nu va putea prevedea conținutul ei. Compilatorul creează *variabile statice* când începe executarea programului. Variabilele statice, de regulă, corespund variabilelor globale. Majoritatea compilatoarelor de C inițializează variabilele statice cu 0. Pe durata executării programului, compilatorul alocă *variabile dinamice* din zona heap. În majoritatea cazurilor, programul trebuie să inițializeze variabilele dinamice.

Observație: *Anumite funcții run-time de bibliotecă vor inițializa cu 0 locațiile de memorie dinamică, în timp ce altele nu.*

FUNCȚII CARE ACCEPTĂ UN NUMĂR VARIABIL DE PARAMETRI

C/C++ 282

Așa cum ați învățat, compilatorul de C înlocuiește parametrii formali, definiți în antetul unei funcții, cu parametrii reali, transmiși funcției. Dacă funcția așteaptă trei parametri, apelarea funcției trebuie să includă trei valori pentru parametrii respectivi. Dacă examinați funcții ca *printf* sau *scanf*, veți observa că ele acceptă un număr variabil de parametri. De exemplu, sunt valabile următoarele apeluri ale funcției *printf*:

```
printf("Totul despre C/C++");
printf(''%d %d %d %d %d'', 1, 2, 3, 4, 5);
printf(''%f %s %s %d %x'', salariu, nume, statut, varsta, id);
```

Așa cum veți învăța în secțiunea 283, puteți folosi macroinstrucțiunile *va_arg*, *va_end*, *va_start* (definite în fișierul antet *stdarg.h*) pentru a crea în programele dumneavoastră funcții cu un număr variabil de parametri. Macroinstrucțiunile, în esență, extrag parametrii din stivă unul câte unul, până când programul ajunge la ultimul parametru. Când folosiți aceste macroinstrucțiuni pentru a obține parametrii, trebuie să știți exact tipul fiecărui parametru. În cazul lui *printf*, funcția utilizează specificatori de format (de exemplu %d, %f, %s) pentru a asocia tipurile parametrilor.

283 ACCEPTAREA UNUI NUMĂR VARIABIL DE PARAMETRI



În această secțiune, veți crea o funcție numită *ad_val*, care adună toate valorile întregi transmise funcției de către funcția apelantă. Cum se vede mai jos, funcția acceptă un număr variabil de parametri. Valoarea 0 din apelul funcției indică ultimul parametru (care nu afectează suma):

```
rezultat = ad_val(3,0);               // Returneaza 3
rezultat = ad_val(3,5,0);             // Returneaza 8
rezultat = ad_val(100,3,4,2,0);       // Returneaza 109
```

Următorul program, *ad_val.c*, conține și folosește funcția *ad_val*:

```
#include <stdio.h>
#include <stdarg.h>

int ad_val(int val, ...)
  {
    va_list argument_ptr;
    int rezultat = 0;
    if (val != 0)
      {
        rezultat += val;
        va_start(argument_ptr, val);
        while ((val = va_arg(argument_ptr, int)) != 0)
          rezultat += val;
        va_end(argument_ptr);
      }
    return(rezultat);
}

void main(void)
  {
    printf("Suma lui 3 e %d\n", ad_val(3, 0));
    printf("Suma lui 3 + 5 e %d\n", ad_val(3, 5, 0));
    printf("Suma lui 3 + 5 + 8 e %d\n", ad_val(3, 5, 8, 0));
    printf("Suma lui 3 + 5 + 8 + 9 e %d\n", ad_val(3, 5, 8, 9, 0));
  }
```

Funcția *ad_val* folosește macroinstrucțiunea *va_start* pentru a atribui unui pointer (*argument_ptr*) adresa primului parametru din stivă. Apoi, funcția folosește macroinstrucțiunea *va_arg* pentru a obține valorile una câte una. Macroinstrucțiunea *va_arg* returnează o valoare de tipul specificat și incrementează *argument_ptr*, astfel încât să indice următorul argument. Când *argument_ptr* întâlnește terminatorul 0, funcția folosește macroinstrucțiunea *va_end* pentru a atribui lui *argument_prt* o valoare care previne folosirea ulterioară a acestui pointer (până când *va_start* îl reinițializează). Când creați funcții care acceptă un număr variabil de parametri, funcțiile dumneavoastră trebuie să aibă o modalitate de a cunoaște numărul de parametrii și tipul fiecăruia. În cazul lui *printf*, specificatorul de format definește parametrii și tipurile lor. În cazul lui *ad_val*, terminatorul 0 marchează ultimul parametru. De asemenea, toate argumentele transmise funcției sunt de același tip.

Observație: *Rețineți utilizarea punctelor de suspensie (...) în cadrul antetului funcției* ***ad_val*** *pentru a indica un număr variabil de parametri.*

FUNCȚIONAREA MACROINSTRUCȚIUNILOR VA_START, VA_ARG ȘI VA_END

C/C++ 284

În secțiunea 283, ați învățat că puteți utiliza macroinstrucțiunile *va_start*, *va_arg* și *va_end*, definite în fișierul antet *stdarg.h*, pentru a crea funcții care acceptă un număr variabil de parametri. Pentru a înțelege mai bine funcționarea acestor macroinstrucțiuni, să analizăm următorul apel al funcției *ad_val*:

```
ad_val(10, 20, 30, 0);
```

Când programul apelează funcția, compilatorul va plasa parametrii în stivă de la dreapta la stânga. În cadrul funcției, macroinstrucțiunea *va_strat* atribuie unui pointer adresa primului parametru, cum se vede în figura 284.

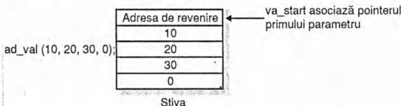


Figura 284 *Utilizarea macroinstrucțiunii* ***va_start*** *pentru a atribui unui pointer adresa primului parametru.*

Macroinstrucțiunea *va_arg* returnează valoarea indicată de pointerul argumentului. Pentru a determina valoarea, macroinstrucțiunea trebuie să cunoască tipul parametrului. Un parametru de tip *int*, de exemplu, va utiliza 16 biți, pe când un parametru de tip *long* va folosi 32 de biți. După regăsirea valorii, macroinstrucțiunea *va_arg* va incrementa pointerul, astfel încât să indice următorul argument din stivă. Pentru a determina numărul de octeți adăugați pointerului, *va_arg* va folosi din nou tipul parametrului. După ce macroinstrucțiunea *va_arg* regăsește ultimul argument, macroinstrucțiunea *va_end* va anula valoarea pointerului argumentului.

CREAREA FUNCȚIILOR CARE ACCEPTĂ PARAMETRI ȘI TIPURI MULTIPLE

C/C++ 285

În secțiunile 282 și 283, ați învățat cum să creați funcții care acceptă un număr variabil de parametri. Din păcate, funcția *ad_val* nu acceptă decât parametri de tip *int*. Următorul program, *tipuri.c*, modifică funcția *ad_val* pentru a accepta valori de toate tipurile. Funcția returnează o valoare de tip *float*. Pentru a ajuta funcția să determine tipul parametrului, i se transmite un specificator de format similar cu cel din *printf*. De exemplu, pentru a aduna trei valori întregi, utilizați următorul model de apel:

```
rezultat = ad_val(''%d %d %d'', 1, 2, 3);
```

La fel, pentru a aduna trei valori reale în virgulă mobilă, utilizați următorul model de apel:

```
rezultat = ad_val(''%f %f %f'', 1.1, 2.2, 3.3);
```

În sfârșit, pentru a aduna valori întregi și în virgulă mobilă, folosiți următorul apel:

```
rezultat = ad_val(''%f %d %f %d'', 1.1, 2, 3.3, 4);
```

Folosind specificatorul de format, eliminați nevoia de a utiliza un terminator 0. În plus, specificatorul de format vă permite să determinați câți biți folosește fiecare parametru, ca în exemplul de mai jos:

```
#include <stdio.h>
#include <stdarg.h>

double ad_val(char *sir, ...)
  {
    va_list marcator;
    double rezultat = 0.0;

    va_start(marcator, sir); // Marcheaza primul argument
                             // aditional
    while (*sir) // Examineaza fiecare caracter din sir
      {
        if (*sir == '%') // Daca nu e specificator de format,
                         // sarim peste
          {
            switch (*(++sir))
              {
                case 'd': rezultat += va_arg(marcator, int);
                 break;
                case 'f': rezultat += va_arg(marcator, double);
                 break;
              }
          }
        sir++;
      }

    va_end(marcator);
    return(rezultat);
  }

void main(void)
  {
    double rezultat;

    printf("Rezultat %f\n", add_val("%f", 3.3));
    printf("Rezultat %f\n", add_val("%f %f", 1.1, 2.2));
    printf("Rezultat %f\n", add_val("%f %d %f", 1.1, 1, 2.2));
    printf("Rezultat %f\n", add_val("%f %d %f %d", 1.1, 1,
      2.2, 3));
  }
```

CITIREA UNUI CARACTER DE LA TASTATURĂ

C/C++ 286

Chiar și cel mai simplu program în C trebuie să citească deseori caractere de la tastatură. Caracterul citit poate avea semnificația unei opțiuni de meniu, a unui răspuns Da/Nu sau a uneia dintre multele litere ale unui nume. Programele execută adesea operații de introducere a caracterelor folosind macroinstrucțiunea *getchar*. Veți implementa macroinstrucțiunea *getchar* ca mai jos:

```
#include <stdio.h>
int getchar(void);
```

Dacă implementarea reușește, macroinstrucțiunea *getchar* returnează valoarea ASCII a caracterului pe care l-a citit. Dacă apare o eroare sau întâlnește sfârșitul de fișier (în mod obișnuit, pentru intrările redirectate), *getchar* returnează *EOF* (End of File – sfârșit de fișier). Următorul program, *getchar.c*, folosește macroinstrucțiunea *getchar* pentru a citi un răspuns Da sau Nu de la tastatură:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
  {
    int litera;

    printf("Introduceti D sau N pentru a continua si apoi
      apasati Enter\n");
    do
      {
       litera = toupper(getchar());
      }
    while ((litera != 'D') && (litera != 'N'));
    printf("Ati introdus %c\n", ((litera == 'D') ? 'D': 'N'));
  }
```

După cum vedeți, programul utilizează bucla *do while* pentru invocarea repetată a macroinstrucțiunii *getchar* până când utilizatorul introduce fie D, fie N.

Observație: *Pentru a accepta redirectarea I/O, limbajul C definește de fapt macroinstrucțiunea* ***getchar*** *cu* ***stdin*** *(care corespunde în mod prestabilit tastaturii).*

AFIȘAREA UNUI CARACTER DE IEȘIRE

C/C++ 287

În secțiunea 286, ați învățat cum se utilizează macroinstrucțiunea *getchar* pentru a citi un caracter de la tastatură. În mod similar, limbajul C pune la dispoziție macroinstrucțiunea *putchar*. Macroinstrucțiunea *putchar* scrie caractere pe ecran (*stdout*). Formatul macroinstrucțiunii *putchar* este următorul:

```
#include <stdio.h>

int putchar(int litera);
```

Dacă execuţia macroinstrucţiunii *putchar* reuşeşte, ea returnează caracterul scris. Dacă apare o eroare, *getchar* returnează *EOF*. Următorul program, *putchar.c*, foloseşte macroinstrucţiunea *getchar* pentru a afişa literele alfabetului:

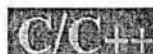
```
#include <stdio.h>

void main(void)
  {
   int litera;

   for (litera = 'A'; litera <= 'Z'; litera++)
    putchar(litera);
  }
```

Observaţie: *Pentru că limbajul C defineşte* ***putchar*** *utilizând* ***stdout****, puteţi utiliza operatorul de redirectare a ieşirii DOS pentru a redirecta ieşirea programului* ***putchar.c*** *spre un fişier sau spre imprimantă.*

288 UTILIZAREA UNUI BUFFER DE INTRARE



Atunci când programele dumneavoastră utilizează un buffer de intrare, sistemul de operare transmite către program litera pe care o introduce utilizatorul doar după apăsarea tastei ENTER. În acest fel, utilizatorul poate să schimbe caracterele pe care le-a introdus, folosind tasta BACKSPACE pentru a şterge caracterele conform necesităţilor. Când utilizatorul apasă tasta ENTER, toate caracterele introduse sunt accesibile programului. Macroinstrucţiunea *getchar* foloseşte un buffer de intrare. Dacă utilizaţi macroinstrucţiunea *getchar* pentru a citi un răspuns alcătuit dintr-un singur caracter, *getchar* nu citeşte caracterul până când utilizatorul nu apasă tasta ENTER. Dacă utilizatorul introduce mai multe caractere, toate caracterele sunt accesibile macroinstrucţiunii *getchar* în cadrul bufferului de intrare. Următorul program, *bufferio.c*, ilustrează utilizarea unui buffer de intrare. Executaţi programul şi apoi introduceţi un rând de text. Caracterele pe care le introduceţi nu vor fi accesibile programului până nu apasaţi tasta ENTER. După ce apasaţi ENTER, programul va citi şi afişa caracterele până va întâlni caracterul *linie nouă* (*newline*), pe care sistemul de operare îl creează când apăsaţi tasta ENTER, aşa cum se vede mai jos:

```
#include <stdio.h>

void main(void)
  {
   int litera;
   do
     {
      litera = getchar();
      putchar(litera);
     }
   while (litera != '\n');
  }
```

Când executaţi programul *bufferio.c*, faceţi încercări cu literele pe care le introduceţi (ştergeţi literele folosind tasta BACKSPACE şi aşa mai departe). Cum veţi vedea, programului îi vor fi transmise literele care corespund textului dumneavoastră final.

ATRIBUIREA INTRĂRII DE LA TASTATURĂ UNUI ȘIR DE CARACTERE

C/C++ 289

Capitolul „Șiruri" prezintă mai multe modalități de gestionare a șirurilor de caractere. Când executați o intrare de la tastatură, una dintre cele mai multe obișnuite operații pe care le poate executa programul dumneavoastră este atribuirea caracterelor rezultate din introducerea unui șir de caractere de la tastatură. Următorul program, *olinie.c*, utilizează macroinstrucțiunea *getchar* pentru a atribui litere unei variabile de tip *șir de caractere*. Pentru a atribui caractere, programul execută pur și simplu o buclă, atribuind caractere elementelor șirului până când întâlnește caracterul *linie nouă*. Apoi, programul atribuie caracterul *NULL* (sfârșitul șirului de caractere) în poziția curentă a șirului, ca mai jos:

```
#include <stdio.h>
void main(void)
  {
    char sir[128];
    int index = 0;
    int litera;
    printf("Introduceti un sir si apasati Enter\n");
    while ((litera = getchar()) != '\n')
      sir[index++] = litera;
    sir[index] = NULL;
    printf("Sirul de caractere a fost: %s\n", sir);
  }
```

COMBINAREA MACROINSTRUCȚIUNILOR GETCHAR ȘI PUTCHAR

C/C++ 290

Așa cum ați învățat, macroinstrucțiunea *getchar* vă permite să citiți o literă de la tastatură (*stdin*), pe când *putchar* vă permite să afișați o literă pe ecran (*stdout*). În funcție de scopul programului dumneavoastră, puteți să citiți și să afișați caractere simultan. Următoarea buclă *do while* va citi și va afișa caractere până la caracterul linie nouă, inclusiv:

```
do
  {
    litera = getchar();
    putchar(litera);
  }
while (litera != '\n');
```

Deoarece ambele macroinstrucțiuni lucrează cu valori întregi, puteți combina instrucțiunile precedente, ca mai jos:

```
do
  putchar(litera = getchar());
while (litera != '\n');
```

În acest caz, *getchar* va atribui caracterul introdus variabilei *litera*, iar macroinstrucțiunea *putchar* va afișa valoarea atribuită.

291 MACROINSTRUCȚIUNILE GETCHAR ȘI PUTCHAR C/C++

Când veți crea programe, nu uitați că *getchar* și *putchar* sunt macroinstrucțiuni, nu funcții. Prin urmare, nu toate compilatoarele vă vor permite să lăsați spații între numele lor și paranteze, cum se vede mai jos:

```
litera = getchar();
putchar(litera);
```

Dacă examinați fișierul antet *stdio.h*, veți întâlni definițiile macroinstrucțiunilor *getchar* și *putchar*. Capitolul „Redirectarea I/O și procesarea liniei de comandă" va explica în detaliu definițiile acestor macroinstrucțiuni.

292 CITIREA UNUI CARACTER UTILIZÂND O INTRARE/IEȘIRE DIRECTĂ C/C++

Ați învățat în secțiunea 288 că, atunci când introduceți date de la tastatură, programele dumneavoastră poate efectua o citire directă sau prin intermediul bufferelor. Atunci când programele dumneavoastră efectuează operații de intrare directă, caracterele introduse de utilizatori sunt imediat disponibile în program (cu alte cuvinte, sistemul de operare nu preia caracterele prin intermediul unui buffer). Dacă utilizatorul apasă tasta BACKSPACE pentru a șterge ulterior un caracter, programul însuși trebuie să realizeze operația de editare (ștergerea caracterului precedent de pe ecran și înlăturarea lui din buffer). Funcția *getche* permite programelor dumneavoastră să citească un caracter de la tastatură utilizând citirea directă. Formatul funcției *getche* este următorul:

```
#include <conio.h>
int getche(void);
```

Următorul program, *getche.c*, utilizează funcția *getche* pentru a citi un răspuns Da sau Nu de la tastatură:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main(void)
  {
    int litera;
    printf("Doriti sa continuati? (D/N): ");
    do
      {
        litera = getche();
        litera = toupper(litera);
      }
    while ((litera != 'D') && (litera != 'N'));
```

```
    if (litera == 'D')
      printf("\nRaspunsul a fost Da\n");
    else
      printf("\nDe ce nu?\n");
}
```

Spre deosebire de programul *getchar.c*, care cere ca utilizatorul să apase tasta ENTER pentru a face răspunsul accesibil, programul *getche.c* oferă acces imediat la caracterele introduse de utilizator.

INTRAREA DIRECTĂ DE LA TASTATURĂ FĂRĂ AFIȘAREA CARACTERELOR

C/C++ 293

În secțiunea 292, ați învățat cum se utilizează funcția *getche* pentru a citi caractere de la tastatură în momentul în care le introduce utilizatorul (folosind intrarea/ieșirea directă). Când utilizați funcția *getche*, programul afișează pe ecran literele introduse de utilizator, pe măsură ce sunt introduse. În funcție de programele dumneavoastră, pot apărea situații în care trebuie să citiți caracterele de la tastatură fără ca ele să apară pe ecran. De exemplu, dacă programul cere utilizatorului o parolă, ar trebui ca literele pe care le introduce acesta să nu apară pe ecran, pentru a nu fi văzute de alții. Funcția *getch* permite ca programele dumneavoastră să citească caractere de la tastatură fără ca ele să fie afișate (în ecou) pe ecran. Formatul funcției *getch* este următorul:

```
#include <conio.h>
int getch(void);
```

Următorul program, *getch.c*, folosește funcția *getch* pentru a citi caractere de la tastatură. Pe măsură ce utilizatorul le introduce, programul utilizează funcția *getch* pentru a citi fiecare caracter, îl convertește în majusculă și apoi afișează pe ecran majuscula echivalentă. Programul *getch.c* arată cum puteți implementa rapid un astfel de proces:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main(void)
  {
    int litera;
    printf("Tastati un sir de caractere si apasati Enter\n");
    do
      {
        litera = getch();
        litera = toupper(litera);
        putch(litera);
      }
    while (litera != '\r');
  }
```

294 UTILIZAREA SECVENȚELOR ESCAPE '\R' ȘI '\N'



Așa cum ați învățat, limbajul C utilizează secveța escape '\r' pentru a indica un retur de car. De asemenea, limbajul C utilizează '\n' pentru a reprezenta o *linie nouă* (retur de car și avans de rând). Când programele preiau intrarea prin buffer utilizând funcția *getchar*, compilatorul de C convertește apăsarea tastei ENTER în secvența retur de car și avans de rând (*linie nouă*). Pe de altă parte, când utilizați o intrare/ieșire directă cu *getche* sau *getch*, fiecare dintre aceste funcții va returna tasta ENTER numai ca secvență de retur de car ('\r'). De aceea, trebuie să verificați corectitudinea caracterului în cadrul programului dumneavoastră, cum se vede mai jos:

```
do
   {
     litera = getchar();
     putch(litera);
   }
while (litera != '\n');
do
   {
     litera = getch();
     putch(litera);
   }
while (litera != '\r');
```

295 EXECUTAREA IEȘIRII DIRECTE



Așa cum ați învățat, funcțiile *getche* și *getch* permit programelor dumneavoastră citirea caracterelor direct de la tastatură, evitând trecerea prin bufferul creat de sistemul de fișiere. În mod similar, programele dumneavoastră pot să execute o afișare rapidă pe ecran folosind funcția *putch*, cum se vede mai jos:

```
#include <conio.h>
int puch(int litera);
```

Dacă se execută cu succes, funcția *putch* returnează litera spre afișare. Dacă a apărut o eroare, funcția *putch* returnează *EOF*. Pentru a executa o ieșire rapidă, funcția *putch* comunică cu serviciile video BIOS sau accesează direct memoria video a PC-ului. Funcții cum ar fi *putchar*, pe de altă parte, utilizează sistemul de fișiere, care, la rândul lui, apelează sistemul BIOS. Funcția *putch* nu convertește caracterul avans de rând în secvența retur de car și avans de rând. Următorul program, *putch.c*, folosește funcțiile *putch* și *putchar* pentru a afișa pe ecran literele alfabetului de 1001 ori. Programul afișează apoi durata de timp pe care o cere executarea fiecărei funcții, cum se vede mai jos:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
   {
```

```
    int litera;
    int nr;

    time_t start_time, stop_time;
    time(&start_time);
    for (nr = 0; nr < 1000; nr++)
      for (litera = 'A'; litera <= 'Z'; litera++)
        putchar(litera);
    time(&stop_time);
    printf("\n\nTimpul cerut de putchar %d secunde\n",
    stop_time-start_time);
    printf("Apasati orice tasta...\n");
    getch();
    time(&start_time);
    for (nr = 0; nr < 1001; nr++)
      for (litera = 'A'; litera <= 'Z'; litera++)
        putch(litera);
    time(&stop_time);
    printf("\n\nTimpul cerut de putch %d secunde\n",
    stop_time-start_time);
  }
```

REINTRODUCEREA ÎN BUFFER A CARACTERELOR

C/C++ 296

Aşa cum aţi învăţat, funcţia *getch* permite programelor dumneavoastră să preia caractere introduse de la tastatură. În funcţie de modul în care vă veţi scrie programul, puteţi să citiţi şi să prelucraţi intrările de la tastatură până la un caracter specificat, apoi să citiţi caracterele rămase. Atunci când scrieţi un astfel de cod, e posibil ca programul dumneavoastră să „anuleze" citirea unui caracter. Funcţia *ungetch* permite programelor să nu citească un caracter. Pentru a face aceasta, veţi implementa funcţia *ungetch*, ca mai jos:

```
#include <conio.h>

int ungetch(int caracter);
```

În plus, puteţi să plasaţi un caracter în bufferul tastaturii, astfel ca programul să îl mai poată citi o dată. Folosind funcţia *ungetch*, programele dumneavoastră vor putea face acest lucru. Următorul program, *ungetch.c*, citeşte literele de la tastatură până când întâlneşte o literă care nu e minusculă. Programul afişează literele citite, apoi citeşte şi afişează pe altă linie caracterele rămase:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main(void)
  {
    int litera;
```

```
    int gata = 0;
    int maj_gasite = 0;

    do
      {
        litera = getch();
        if (islower(litera))
          putchar(litera);
        else
          {
            if (isupper(litera))
              {
                ungetch(litera);
                maj_gasite = 1;
                putchar('\n');
              }
            gata = 1;
          }
      }
    while (! gata);
    if (majusc_gasit)
      do
        {
          litera = getch();
          putchar(litera);
        }
      while (litera != '\r');
}
```

Dacă citiți caracterele folosind funcția *getchar*, puteți folosi funcția *ungetc* pentru a anula citirea unui caracter, ca mai jos:

```
ungetc(litera, stdin);
```

297 FORMATAREA RAPIDĂ A IEȘIRILOR CU CPRINTF C/C++

După cum știți, funcția *printf* permite programelor dumneavoastră să creeze ieșiri formatate. Compilatorul de C definește, de fapt, funcția *printf* utilizând indicatorul de fișier *stdout*. Ca urmare, puteți redirecta ieșirea creată cu *printf* astfel încât să nu mai fie trimisă la ecran, ci spre un fișier sau un dispozitiv. Pentru că funcția *printf* afișează caracterele cu *stdout*, folosește sistemul de fișiere al limbajului C, care la rândul lui, folosește funcțiile DOS. Fiecare dintre funcțiile DOS, la rândul lor, apelează la BIOS. Pentru a formata mai rapid o ieșire, programul dumneavoastră poate utiliza următoarea funcție, *cprintf*, care lucrează direct cu sistemul BIOS sau cu memoria video a calculatorului:

```
#include <conio.h>

int cprintf(const char *format[,argumente...]);
```

Următorul program, *cprintf.c*, scrie pe ecranul dumneavoastră de 1001 ori șirul de caractere *„Totul despre C/C++"* folosind funcția *printf* și apoi funcția *cprintf*. Programul afișează apoi duratele de timp cerute de cele două funcții:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
  {
    int nr;
    time_t start_time, stop_time;
    time(&start_time);
    for (nr = 0; nr < 1001; nr++)
      printf("Totul despre C/C++\n");
    time(&stop_time);
    printf("\n\nTimpul cerut de printf %d secunde\n",
      stop_time-start_time);
    printf("Apasati orice tasta...\n");
    getch();
    time(&start_time);
    for (nr = 0; nr < 1001; nr++)
      cprintf("Totul despre C/C++\r\n");
    time(&stop_time);
    printf("\n\nTimpul cerut de cprintf %d secunde\n",
      stop_time-start_time);
  }
```

Observație: *Funcția* ***cprintf*** *nu convertește caracterul* ***linie nouă*** *în secvența retur de car și avans de rând.*

FORMATAREA RAPIDĂ A INTRĂRILOR DE LA TASTATURĂ

C/C++ 298

În secțiunea 297, ați învățat că funcția *cprintf* permite programelor dumneavoastră să evite sistemul de fișiere pentru a afișa mai rapid ieșirile pe ecran. În mod similar, funcția *cscanf* permite programelor dumneavoastră să execute formatarea rapidă a intrărilor de la tastatură, ca mai jos:

```
#include <conio.h>
int cscanf(char *format[,argumente]);
```

Următorul program, *cscanf.c*, vă cere să introduceți trei valori întregi. Programul citește apoi valorile folosind funcția *cscanf*:

```
#include <conio.h>

void main(void)
  {
    int a, b, c;
```

```
    cprintf("Introduceti 3 valori intregi si apasati Enter\r\n");
    cscanf("%d %d %d", &a, &b, &c);
    cprintf("Valorile introduse %d %d %d\r\n", a, b, c);
  }
```

299 SCRIEREA UNUI ŞIR DE CARACTERE

C/C++

Aşa cum aţi învăţat, funcţia *printf* permite programelor dumneavoastră să afişeze pe ecran ieşiri formatate. Folosind funcţia *printf*, programele dumneavoastră pot să scrie pe ecran şiruri de caractere, valori întregi, valori reale în virgulă mobilă sau combinaţii de valori diferite. Când programul dumneavoastră are însă nevoie să scrie numai şiruri de caractere, aveţi posibilitatea să îmbunătăţiţi performanţele programului folosind funcţia *puts* în locul funcţiei *printf*, ca mai jos:

```
#include <stdio.h>

int puts(const char *sir);
```

Funcţia *puts* scrie pe ecran un şir de caractere terminat cu *NULL*. Dacă funcţia *puts* se execută cu succes, ea returnează o valoare nenegativă. Dacă apare o eroare, funcţia *puts* returnează *EOF*. Funcţia *puts* scrie automat caracterul *linie nouă* la sfârşitul şirului de caractere. Următorul program, *puts.c*, foloseşte funcţiile *prinf* şi *puts* pentru a afişa de 1001 ori şirul de caractere *„Totul despre C/C++"*. Programul afişează perioada de timp cerută de fiecare funcţie:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
  {
    int nr;
    time_t start_time, stop_time;
    time(&start_time);
    for (nr = 0; nr < 1001; nr++)
     printf("Totul despre C/C++\n");
    time(&stop_time);
    printf("\n\nTimpul cerut de printf %d secunde\n",
    stop_time-start_time);
    printf("Apasati orice tasta...\n");
    getch();
    time(&start_time);
    for (nr = 0; nr < 1001; nr++)
     puts("Totul despre C/C++");
    time(&stop_time);
    printf("\n\nTimpul cerut de puts %d secunde\n",
    stop_time-start_time);
  }
```

***Observație:** Pentru că funcția **puts** adaugă automat la sfârșitul șirului caracterul **linie nouă**, șirul de caractere pe care programul îl cere funcției **puts** să-l afișeze nu trebuie să conțină caracterul linie nouă.*

IEȘIREA MAI RAPIDĂ A UNUI ȘIR DE CARACTERE FOLOSIND O INTRARE/IEȘIRE DIRECTĂ

C/C++ 300

În secțiunea 299, ați învățat că funcția *puts* permite programelor dumneavoastră să afișeze rapid un șir de caractere. Însă, deoarece funcția *puts* este definită cu *stdout* (pentru a accepta redirectarea), trebuie să folosească sistemul de fișiere. Pentru a afișa mai rapid un șir de caractere pe ecran, programul poate folosi funcția *cputs*, ca mai jos:

```
#include <conio.h>

int cputs(const char *sir);
```

La fel ca și *puts*, funcția *cputs* afișează un șir de caractere terminat cu *NULL*. Însă, spre deosebire de *puts*, funcția *cputs* nu adaugă automat la sfârșitul șirului caracterul *linie nouă*. Următorul program, *cputs.c*, folosește funcțiile *puts* și *cputs* pentru a afișa de 1500 de ori șirul de caractere *„Totul despre C/C++"*. Programul afișează apoi perioada de timp cerută de fiecare dintre cele două funcții pentru a genera această ieșire:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
 {
   int nr;
   time_t start_time, stop_time;
   time(&start_time);
   for (nr = 0; nr < 1500; nr++)
     puts("Totul despre C/C++");
   time(&stop_time);
   printf("\n\nTimpul cerut de puts %d secunde\n",
   stop_time-start_time);
   printf("Apasati orice tasta...\n");
   getch();
   time(&start_time);
   for (nr = 0; nr < 1500; nr++)
     cputs("Totul despre C/C++\r\n");
   time(&stop_time);
   printf("\n\nTimul cerut de cputs %d secunde\n",
   stop_time-start_time);
 }
```

301 CITIREA ȘIRURILOR DE CARACTERE DE LA TASTATURĂ



În secțiunea 299, ați învățat că limbajul C furnizează funcția *puts* pentru a scrie un șir de caractere pe ecran. În mod similar, programele dumneavoastră pot folosi funcția *gets* pentru a citi un șir de caractere intodus de la tastatură, ca mai jos:

```
#include <stdio.h>
char *gets(char *sir);
```

Dacă funcția se execută cu succes, va returna un pointer la șirul de caractere. Dacă se produce o eroare sau întâlnește simbolul de sfârșit de fișier, returnează valoarea *NULL*. Funcția *gets* citește caracterele până la caracterul *linie nouă*, inclusiv. Însă funcția *gets* înlocuiește caracterul *linie nouă* cu caracterul *NULL*. Următorul program, *gets.c*, folosește funcția *gets* pentru a citi un șir de caractere de la tastatură:

```
#include <stdio.h>
void main(void)
  {
    char sir[256];
    printf("Introduceti un sir de caractere si apasati Enter\n");
    gets(sir);
    printf("Sirul a fost %s\n", sir);
  }
```

***Observație:** Limbajul C definește, de fapt, funcția **gets** utilizând **stdin** (care este în mod prestabilit tastatura), astfel încât este posibilă redirectarea I/O.*

302 INTRODUCEREA RAPIDĂ A UNUI ȘIR DE CARACTERE DE LA TASTATURĂ



În secțiunea 301, ați învățat cum puteți folosi în programele dumneavoastră funcția *gets* pentru a citi un șir de caractere introdus de la tastatură. Pentru că limbajul C definește *gets* utilizând *stdin*, această funcție trebuie să utilizeze sistemul de fișiere pentru a realiza operația de intrare. Dacă nu aveți nevoie de redirectarea I/O, puteți să citiți un șir de caractere de la tastatură cu funcția *cgets*, ceea ce va mări performanța programului dumneavoastră. Veți implementa funcția *cgets* astfel:

```
#include <stdio.h>
char *cgets(char *sir);
```

Dacă funcția *cgets* reușește să realizeze citirea șirului de caractere de la tastatură, va returna un pointer care va indica locația *sir[2]* din șirul de caractere. Dacă apare o eroare, funcția returnează *NULL*. Funcția *cgets* funcționează diferit față de funcția *gets*. Înainte de a apela funcția *cgets* cu un șir de caractere, trebuie să introduceți în locația *sir[0]* numărul maxim de caractere pe care funcția *cgets* îl va citi. La revenire, *sir[1]* va conține numărul caracterelor citite de funcția *cgets*. Șirul de caractere terminat cu *NULL* începe, de fapt, la *sir[2]*. Următorul program, *cgets.c*, ilustrează modul de utilizare a funcției *cgets*:

```
#include <stdio.h>
#include <conio.h>

void main(void)
  {
    char buffer[256];
    buffer[0] = 253; // Numarul de caractere care pot fi citite
    printf("Introduceti un sir si apasati Enter\n");
    cgets(buffer);
    printf("\n\nNumarul de caractere citite este %d\n", buffer[1]);
    printf("Sirul citit: %s\n", &buffer[2]);
  }
```

Ca un experiment, reduceți la 10 numărul de caractere pe care funcția *cgets* le poate citi. Dacă utilizatorul încearcă să introducă mai mult decât 10 caractere, funcția va ignora caracterele suplimentare.

AFIȘAREA IEȘIRII ÎN CULORI

C/C++ 303

Utilizând driverul de dispozitiv *ansi.sys*, programele dumneavoastră pot afișa pe ecran o ieșire în culori. În plus, multe compilatoare de C furnizează funcții specializate pentru ieșiri de tip text care permit afișarea în culori. Dacă utilizați *Turbo C++ Lite*, Borland C++ sau Microsoft C++, funcția *outtext* (numită *_outtext* în Microsoft C++) vă permite afișarea unei ieșiri în culori. Dacă utilizați *Turbo C++ Lite* sau Borland C++, puteți folosi funcția *outtext* numai în modul grafic. Funcția *_outtext* din Microsoft C++, pe de altă parte, lucrează atât în modul grafic, cât și în modul de text. Dacă trebuie să realizați o ieșire în culori, studiați documentația care însoțește compilatorul dumneavoastră, pentru precizări legate de aceste funcții. După cum veți vedea, compilatorul dispune de funcții care stabilesc poziția textului, culorile și modurile grafice. Pentru că rutinele de ieșire ANSI depind de compilator, această carte nu se ocupă de ele.

ȘTERGEREA ECRANULUI

C/C++ 304

Cele mai multe compilatoare de C nu oferă funcții care să permită ștergerea ecranului. Dacă folosiți *Turbo C++ Lite*, Borland C++ sau Microsoft C++, puteți totuși să utilizați funcția *clrscr* pentru a șterge conținutul unei ferestre în modul de text, cum se vede mai jos:

```
#include <conio.h>

void clrscr(void);
```

Următorul program, *clrscr.c*, folosește funcția *clrscr* pentru ștergerea ecranului:

```
#include <conio.h>

void main(void)
  {
    clrscr();
  }
```

305 ȘTERGEREA PÂNĂ LA SFÂRȘITUL LINIEI CURENTE

C/C++

Când veți crea programe ale căror ieșiri sunt afișate pe ecran, puteți să ștergeți conținutul unei linii de la poziția curentă a cursorului până la sfârșitul liniei. Pentru a realiza aceasta, programele dumneavoastră pot utiliza funcția *clreol*, ca mai jos:

```
#include <conio.h>

void clreol(void);
```

Funcția *clreol* șterge conținutul liniei din dreapta cursorului fără a deplasa cursorul.

306 ȘTERGEREA LINIEI CURENTE

C/C++

Când veți crea programe ale căror ieșiri sunt afișate pe ecran, puteți să ștergeți conținutul unei linii și să mutați restul ieșirii cu un rând mai sus. În acest caz, programele dumneavoastră pot utiliza funcția *delline*, cum se vede mai jos:

```
#include <conio.h>

void delline(void);
```

Următorul program, *delline.c*, umple ecranul cu 24 de linii de text. Când apăsați tasta ENTER, programul va utiliza funcția *delline* pentru a șterge liniile 12, 13 și 14, cum se vede mai jos.

```
#include <conio.h>

void main(void)
  {
    int linie;

    clrscr();
    for (linie = 1; linie < 25; linie++)
      cprintf("Aceasta este linia %d\r\n", linie);
    cprintf("Apasati o tasta pentru a continua: ");
    getch();
    gotoxy(1, 12);
    for (linie = 12; linie < 15; linie++)
      dellinie();
    gotoxy(1, 25);
  }
```

307 POZIȚIONAREA CURSORULUI PENTRU IEȘIREA PE ECRAN

C/C++

Așa cum ați învățat, puteți utiliza driverul de dispozitiv *ansi.sys* pentru a poziționa cursorul pentru operațiile de ieșire pe ecran. Dacă lucrați în mediu DOS, majoritatea compilatoarelor de C vă furnizează funcția *gotoxy*, care permite poziționarea cursorului la intersecția unei coloane cu un rând, ca mai jos:

```
#include <conio.h>
void gotoxy(int coloana, int rand);
```

Parametrul *coloana* precizează poziția coloanei (*x*), de la 1 la 80. Parametrul *rand* precizează poziția rândului (*y*), de la 1 la 25. Dacă una dintre valori este inacceptabilă, compilatorul va ignora funcția *gotoxy*. Următorul program, *gotoxy.c*, folosește funcția *gotoxy* pentru a afișa pe ecran o ieșire într-o anumită poziție:

```
#include <conio.h>
void main(void)
  {
    clrscr();
    gotoxy(1, 5);
    cprintf("Iesire la rand 5 coloana 1\n");
    gotoxy(20, 10);
    cprintf("Iesire la rand 10 coloana 20\n");
  }
```

DETERMINAREA POZIȚIEI RÂNDULUI ȘI A COLOANEI C/C++ 308

În secțiunea 307, ați învățat cum se utilizează funcția *gotoxy* pentru a plasa cursorul într-o anumită poziție pe rânduri și coloane. În multe cazuri, va fi necesar ca programele dumneavoastră să cunoască poziția curentă a cursorului înainte de a executa o operație I/O. Funcțiile *wherex* și *wherey* returnează poziția cursorului pe rânduri și coloane, ca mai jos:

```
#include <conio.h>
int wherex(void);
int wherey(void);
```

Următorul program, *wherexy.c*, eliberează ecranul, scrie trei linii și apoi folosește funcțiile *wherex* și *wherey* pentru a determina poziția curentă a cursorului:

```
#include <conio.h>
void main(void)
  {
    int rand, coloana;
    clrscr();
    cprintf("Aceasta este linia 1\r\n");
    cprintf("Linia 2 este putin mai lunga\r\n");
    cprintf("Aceasta este ultima linie");
    rand = wherey();
    coloana = wherex();
    cprintf("\r\nPozitia cursorului rand %d coloana %d\n",
      rand, coloana);
  }
```

309 INSERAREA UNEI LINII GOALE PE ECRAN



Când veți crea programe ale căror ieșiri sunt afișate pe ecran, puteți să inserați o linie goală pe ecran, astfel încât să puteți introduce un text în mijlocul unui text existent. Pentru a executa această operație, programele dumneavoastră pot utiliza funcția *insline*, ca mai jos:

```
#include <conio.h>

void insline(void);
```

Atunci când invocați funcția *insline*, întregul text de sub poziția curentă a cursorului se mută mai jos cu o linie. Ultima linie de jos a ecranului va trece în afara ferestrei. Următorul program, *insline.c*, scrie 25 de linii de text pe ecran. Apoi, programul utilizează funcția *insline* pentru a insera un text pe linia 12, cum se vede mai jos:

```
#include <conio.h>

void main(void)
  {
    int linie;

    clrscr();
    for (linie = 1; linie < 25; linie++)
      cprintf("Aceasta este linia %d\r\n", linie);
    cprintf("Apasati o tasta pentru a continua: ");
    getch();
    gotoxy(1, 12);
    insline();
    cprintf("Acesta este noul text!!!");
    gotoxy(1, 25);
  }
```

310 COPIEREA TEXTULUI DE PE ECRAN ÎNTR-UN BUFFER



Când creați programe ale căror ieșiri sunt afișate pe ecran, pot apărea situații în care trebuie să fie copiat într-un buffer conținutul curent al ecranului. Pentru a copia un text de pe ecran, programele dumneavoastră pot utiliza funcția *gettext*, ca mai jos:

```
#include <conio.h>

int gettext(int st, int sus, int dr, int jos, void *buffer);
```

Parametrii *st* și *sus* precizează coloana și rândul colțului din stânga-sus al regiunii de ecran pe care doriți să o copiați. La fel, parametrii *dr și jos* precizează colțul din dreapta-jos al acelei regiuni. Funcția *gettext* plasează textul și atributele sale în parametrul *buffer*. PC-ul utilizează un octet de atribut pentru fiecare literă a textului pe care îl afișează pe ecran. Dacă doriți să treceți în buffer 10 caractere, de exemplu, bufferul trebuie să fie suficient de mare pentru a păstra 10 caractere ASCII plus 10 octeți (o lungime de 20 de octeți). Următorul program, *savescr.c*, salvează conținutul unui ecran în modul de text într-un fișier numit *savescr.dat*:

```
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

void main(void)
  {
    char buffer[8000];
    int indicator;

    if ((indicator = creat("SAVESCR.DAT", S_IWRITE)) == -1)
      cprintf("Eroare la deschiderea SAVESCRN.DAT\r\n");
    else
      {
        gettext(1, 1, 80, 25, buffer);
        write(indicator, buffer, sizeof(buffer));
        close(indicator);
      }
  }
```

Observație: *În cele mai multe cazuri, atributul textului curent este 7. Dacă încercați să afișați conținutul fișierului* ***savescr.dat*** *folosind comanda* ***TYPE****, sistemul dumneavoastră va emite un sunet pentru fiecare valoare de atribut.*

SCRIEREA UNUI TEXT DIN BUFFER ÎNTR-O ANUMITĂ POZIȚIE DE PE ECRAN

C/C++ 311

Așa cum ați învățat, multe compilatoare bazate pe mediul DOS furnizează funcții pe care programele dumneavoastră le pot utiliza pentru a controla ieșirea video. În secțiunea 310, ați învățat că programele dumneavoastră pot folosi funcția *gettext* pentru a copia într-un buffer caracterele (și atributele lor) dintr-o zonă a ecranului. După ce ați copiat textul în buffer, puteți să copiați mai târziu din nou textul pe ecran, folosind funcția *puttext*, ca mai jos:

```
#include <conio.h>

int puttext(int st, int sus, int dr, int jos, void *buffer);
```

Parametrii *st*, *sus*, *dr* și *jos* specifică locația ecranului unde doriți să fie scris conținutul bufferului. Parametrul *buffer* conține caracterele și atributele lor, pe care funcția *gettext* le-a depozitat anterior. Următorul program, *puttext.c*, mută textul *„Totul despre C/C++"* pe suprafața ecranului până când apăsați o tastă oarecare:

```
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
  {
```

```
    char buffer[128];
    int rand, coloana;

    clrscr();
    cprintf("Totul despre C/C++\r\n");
    gettext(1, 1, 23, 1, buffer);

    while (! kbhit())
      {
        clrscr();
        rand = 1 + random(24);
        coloana = 1 + random(58);
        puttext(coloana, rand, coloana+22, rand, buffer);
        delay(2000);
      }
  }
```

312 DETERMINAREA PARAMETRILOR MODULUI DE TEXT



Așa cum ați învățat, cele mai multe compilatoare de C vă pun la dispoziție mai multe funcții specializate pentru texte, cu care programele dumneavoastră pot să controleze operațiile de ieșire pe ecran. Pentru a determina parametrii curenți ai ecranului, programele dumneavoastră pot utiliza funcția *gettextinfo*, ca mai jos:

```
#include <conio.h>

void gettextinfo(struct text_info *date);
```

Parametrul *date* este un pointer la o structură de tip *text_info*, cum se vede mai jos:

```
struct text_info
  {
    unsigned char winleft;          // Coloana stanga
    unsigned char wintop;           // Randul de sus
    unsigned char winright;         // Coloana dreapta
    unsigned char winbottom;        // Randul de jos
    unsigned char attribute;        // Atribut text
    unsigned char normattr;         // Atribut normal
    unsigned char currmode;         // Modul de text curent
    unsigned char screenheight;     // Randuri
    unsigned char screenwidth;      // Coloane
    unsigned char curx;             // Cursor coloane
    unsigned char cury;             // Cursor randuri
  };
```

Următorul program, *textinfo.c*, folosește funcția *gettextinfo* pentru a afișa parametrii curenți ai textului:

```
#include <conio.h>
void main(void)
  {
    struct text_info text;
    gettextinfo(&text);
    cprintf("Coordonate ecran %d,%d to %d,%d\r\n",
            text.wintop,text.winleft,text.winbottom,text.winright);
    cprintf("Atribut text %d Atribut normal %d\r\n",
            text.attribute, text.normattr);
    cprintf("Inaltime ecran %d latime %d\r\n",
    text.screenheight,text.screenwidth);
    cprintf("Pozitie cursor rand %d coloana %d\r\n",
    text.cury, text.curx);
  }
```

CONTROLUL CULORILOR ECRANULUI

C/C++ 313

Așa cum ați învățat, programele dumneavoastră pot folosi driverul de dispozitiv *ansi.sys* pentru a afișa pe ecran o ieșire în culori. În plus, multe compilatoare din mediul DOS dispun de funcția *textattr*, care vă permite să selectați culorile de fundal și de prim plan ale textului:

```
#include <conio.h>
void textattr(int atribut);
```

Parametrul *atribut* conține opt biți care precizează culoarea dorită. Cei mai puțin semnificativi patru biți precizează culoarea de prim plan. Următorii trei biți precizează culoarea de fundal, iar cel mai semnificativ bit controlează pâlpâirea. Pentru a selecta o culoare, trebuie să atribuiți biților respectivi valoarea corespunzătoare. Tabelul 313 precizează valorile culorilor.

Culoare	Constantă culoare	Valoare	Utilizare
Negru	*BLACK*	0	Prim plan/fundal
Albastru	*BLUE*	1	Prim plan/fundal
Verde	*GREEN*	2	Prim plan/fundal
Cyan	*CYAN*	3	Prim plan/fundal
Roșu	*RED*	4	Prim plan/fundal
Magenta	*MAGENTA*	5	Prim plan/fundal
Maro	*BROWN*	6	Prim plan/fundal
Gri deschis	*LIGHTGRAY*	7	Prim plan/fundal
Gri închis	*DARKGRAY*	8	Prim plan
Albastru deschis	*LIGHTBLUE*	9	Prim plan
Verde deschis	*LIGHTGREEN*	10	Prim plan

(continuare)

Culoare	Constantă culoare	Valoare	Utilizare
Cyan deschis	*LIGHTCYAN*	11	Prim plan
Roșu deschis	*LIGHTRED*	12	Prim plan
Magenta deschis	*LIGHTMAGENTA*	13	Prim plan
Galben	*YELLOW*	14	Prim plan
Alb	*WHITE*	15	Prim plan
Pâlpâirea	*BLINK*	128	Prim plan

Tabelul 313 *Valorile parametrului atribut pentru culoare.*

Următorul program, *textatr.c*, prezintă culorile de prim plan disponibile:

```
#include <conio.h>

void main(void)
  {
    int culoare;

    for (culoare = 1; culoare < 16; culoare++)
      {
        textattr(culoare);
        cprintf("Aceasta este culoarea %d\r\n", culoare);
      }
    textattr(128 + 15);
    cprintf("Aceasta este palpairea\r\n");
  }
```

314 ATRIBUIREA CULORII DE FUNDAL



Așa cum ați învățat în secțiunea 313, funcția *textattr* permite programelor dumneavoastră să selecteze culorile de prim plan și de fundal. Pentru a stabili culoarea de fundal cu ajutorul funcției *textattr*, programul dumneavoastră trebuie să atribuie valoarea culorii pe care o doriți biților de la 4 la 6 ai valorii culorii. Pentru a atribui valoarea culorii, programul poate utiliza operația de deplasare pe biți sau puteți să declarați o structură cu câmpuri de biți, ca mai jos:

```
struct TextColor {
  unsigned char primplan:4;
  unsigned char fundal:3;
  unsigned char palpaire:1;
  };
```

Următorul program, *setback.c*, folosește structura *TextColor* pentru a stabili culorile ecranului:

```
#include <conio.h>

void main(void)
{
```

```
   union TextColor
     {
       struct
       {
         unsigned char primplan:4;
         unsigned char fundal:3;
         unsigned char palpaire:1;
       } biti_culori;
       unsigned char valoare;
     } culori;
   culori.biti_culori.primplan = BLUE;
   culori.biti_culori.fundal = RED;
   culori.biti_culori.clipire = 1;
   textattr(culori.valoare);
   clrscr();
   cprintf("Acesta este noul text color\n");
}
```

STABILIREA CULORII DE PRIM PLAN UTILIZÂND TEXTCOLOR

C/C++315

Așa cum ați învățat, cele mai multe compilatoare pentru mediul DOS oferă funcția *textattr* pentru selectarea culorilor dorite de fundal și de prim plan. Pentru a simplifica procesul de atribuire a culorilor de prim plan, puteți utiliza funcția *textcolor*, cum se vede mai jos:

```
#include <conio.h>

void textcolor(int culoare_primplan);
```

Parametrul *culoare_primplan* va specifica una dintre valorile culorilor prezentate în tabelul 315.

Culoare	Constantă culoare	Valoare
Negru	*BLACK*	0
Albastru	*BLUE*	1
Verde	*GREEN*	2
Cyan	*CYAN*	3
Roșu	*RED*	4
Magenta	*MAGENTA*	5
Maro	*BROWN*	6
Gri deschis	*LIGHTGRAY*	7
Gri închis	*DARKGRAY*	8
Albastru deschis	*LIGHTBLUE*	9
Verde deschis	*LIGHTGREEN*	10

(continuare)

Culoare	Constantă culoare	Valoare
Cyan deschis	*LIGHTCYAN*	11
Roşu deschis	*LIGHTRED*	12
Magenta deschis	*LIGHTMAGENTA*	13
Galben	*YELLOW*	14
Alb	*WHITE*	15
Palpaire	*BLINK*	128

Tabelul 315 *Valorile culorilor de prim plan acceptate de funcţia* **textcolor.**

Următorul program, *txtcolor.c*, ilustrează utilizarea funcţiei *textcolor* pentru a stabili culorile de prim plan:

```
#include <conio.h>
void main(void)
  {
    int culoare;
    for (culoare = 1; culoare < 16; culoare ++)
      {
        textcolor(culoare);
        cprintf("Aceasta este culoarea %d\r\n", culoare);
      }
    textcolor(128 + 15);
    cprintf("Aceasta este palpairea\r\n");
  }
```

316 STABILIREA CULORII DE FUNDAL UTILIZÂND TEXTBACKGROUND



Aşa cum aţi învăţat, cele mai multe compilatoare pentru mediul DOS oferă funcţia *textattr* pentru selectarea culorilor dorite de fundal şi de prim plan. Pentru a simplifica procesul de atribuire a culorilor de fundal, puteţi să utilizaţi funcţia *textbackground*, cum se vede mai jos:

```
#include <conio.h>
void textbackground(int culoare_fundal);
```

Parametrul *culoare_fundal* trebuie să precizeze una dintre valorile culorilor prezentate în tabelul 316.

Culoare	Constantă culoare	Valoare
Negru	*BLACK*	0
Albastru	*BLUE*	1
Verde	*GREEN*	2
Cyan	*CYAN*	3

Culoare	Constantă culoare	Valoare
Roșu	*RED*	4
Magenta	*MAGENTA*	5
Maro	*BROWN*	6
Gri deschis	*LIGHTGRAY*	7

Tabelul 316 *Valorile acceptate ale culorilor de fundal.*

Următorul program, *backgr.c*, utilizează funcția *textbackground* pentru a afișa diferite culori de fundal:

```
#include <conio.h>
void main(void)
  {
    int culoare;

    for (culoare = 0; culoare < 8; culoare ++)
      {
        textbackground(culoare);
        cprintf("Aceasta este culoarea %d\r\n", culoare);
        cprintf("Apasati orice tasta pentru a continua\r\n");
        getch();
      }
  }
```

CONTROLUL INTENSITĂȚII TEXTULUI

C/C++317

Așa cum ați învățat, cele mai multe compilatoare pentru mediul DOS dispun de funcții pentru a permite controlul ieșirii pe ecran. Când folosiți aceste funcții pentru a scrie un text pe ecran, puteți să controlați intensitatea (luminozitatea) informațiilor pe care le afișează programul. Pentru a modifica intensitatea textului, puteți utiliza una dintre următoarele trei funcții:

```
#include <conio.h>

void highvideo(void);
void lowvideo(void);
void normvideo(void);
```

Funcțiile controlează intensitatea cu care este afișat textul pe ecran. Următorul program, *intensit.c*, ilustrează modul în care se utilizează aceste trei funcții:

```
#include <conio.h>

void main(void)
  {
    clrscr();
    highvideo();
    cprintf("Acest text este in high video\r\n");
    lowvideo();
```

```
    cprintf("Acest text este in low video\r\n");
    normvideo();
    cprintf("Acest text este in normal video\r\n");
}
```

318 DETERMINAREA MODULUI DE TEXT CURENT



Așa cum ați învățat, cele mai multe compilatoare pentru mediul DOS dispun de funcții care pot fi folosite de programele dumneavoastră pentru a permite controlul ieșirii pe ecran. Când programele dumneavoastră realizează o ieșire pe ecran, ele trebuie să cunoască și, eventual, să schimbe modul de text curent al PC-ului. De exemplu, un program care așteaptă 80 de coloane nu va afișa corect rezultatul pe un ecran cu 40 de coloane. Pentru a ajuta programele să schimbe modul de text curent, poate fi utilizată funcția *textmode*, cum se vede mai jos:

```
#include <conio.h>
void textmode(int mod_dorit);
```

Parametrul *mod_dorit* precizează modul de text pe cate îl doriți. Tabelul 318 prezintă modurile de text acceptate.

Constantă	Valoare	Mod de text
LASTMODE	–1	Modul anterior
BW40	0	40 de coloane alb-negru
C40	1	40 de coloane color
BW80	2	80 de coloane alb-negru
C80	3	80 de coloane color
MONO	7	80 de coloane monocrom
C4350	64	43 de linii EGA sau 50 de linii VGA

Tabelul 318 *Modificările valide ale modului de text.*

Următoarea instrucțiune, de exemplu, va selecta modul cu 43 de linii pe un monitor EGA sau 50 de linii pe un monitor VGA:

```
textmode(C4350);
```

Observație: *Dacă folosiți funcția* ***textmode*** *pentru a schimba modul de text curent, modificarea va rămâne efectivă după ce programul va încheia execuția.*

319 DEPLASAREA TEXTULUI DE PE ECRAN DE LA O LOCAȚIE LA ALTA



Așa cum ați învățat, cele mai multe compilatoare pentru mediul DOS dispun de funcții care vă permit să controlați ieșirea de text pe ecran. Dacă programul dumneavoastră realizează frecvent ieșiri pe ecran, este posibil să copiați sau să deplasați un text dintr-o parte a ecranului în alta. Pentru a copia un text de pe ecran, programele dumneavoastră pot utiliza funcția *movetext*, cum se vede mai jos:

```
#include <conio.h>

int movetext(int st, int sus, int dr, int jos,
    int destinatie_st, int destinatie_sus);
```

Parametrii *st*, *sus*, *dr* și *jos* descriu caseta care cuprinde zona textului pe care vreți să-l deplasați. Parametrii *destinatie_st* și *destinatie_sus* precizează locația dorită a colțului din stânga-sus al casetei. Următorul program, *movetext.c*, scrie cinci linii de text pe ecran, apoi vă cere să apăsați o tastă. Când faceți acest lucru, programul va copia textul într-o nouă locație, cum se vede mai jos:

```
#include <conio.h>

void main(void)
  {
    int i;

    clrscr();
    for (i = 1; i <= 5; i++)
      cprintf("Aceasta este linia %d\r\n", i);
    cprintf("Apasati orice tasta\n\r");
    getch();
    movetext(1, 1, 30, 6, 45, 18);
    gotoxy(1, 24);
  }
```

Pentru a deplasa textul în noua locație, spre deosebire de copiere, trebuie să ștergeți textul original după ce programul termină operația *movetext*.

DEFINIREA UNEI FERESTRE DE TEXT

C/C++ 320

Așa cum ați învățat, cele mai multe compilatoare pentru mediul DOS dispun de funcții pe care programele dumneavoastră le pot folosi pentru a controla mai bine ieșirea pe ecran. În mod prestabilit, aceste funcții scriu ieșirea pe întregul ecran, dar puteți să restricționați ieșirea la o anumită zonă a ecranului. Pentru a face aceasta, programul dumneavoastră poate utiliza funcția *window*, ca mai jos:

```
#include <conio.h>

void window(int st, int sus, int dr, int jos);
```

Parametrii *st*, *sus*, *dr* și *jos* definesc colțurile din stânga-sus și din dreapta-jos ale zonei de ecran în interiorul căreia doriți să fie scrisă ieșirea. Următorul program, *window.c*, reduce ieșirea programului la sfertul din stânga-sus al ecranului:

```
#include <conio.h>

void main(void)
  {
    int i, j;

    window(1, 1, 40, 12);
    for (i = 0; i < 15; i++)
```

```
    {
      for (j = 0; j < 50; j++)
        cprintf("%d", j);
      cprintf("\r\n");
    }
  }
```

Când ieşirea programului ajunge la marginea ferestrei, rândul este întrerupt şi se trece pe linia următoare. După ce programul se încheie, operaţiile de ieşire vor avea acces la întregul ecran.

321 FOLOSIREA VALORII ABSOLUTE A UNEI EXPRESII DE TIP ÎNTREG



Valoarea absolută precizează modulul, deci distanţa valorii faţă de 0. Valoarea absolută este întotdeauna pozitivă. De exemplu, valoarea absolută a lui 5 este 5, iar a lui –5 este 5. Petru a ajuta programele dumneavoastră să determine valoarea absolută, limbajul C pune la dispoziţie funcţia *abs*. Funcţia *abs* returnează valoarea absolută a unei expresii de tip întreg. Veţi utiliza funcţia *abs* cum se vede mai jos:

```
#include <stdlib.h>

int abs(int expresie);
```

Următorul program, *vezi_abs.c*, ilustrează modul de utilizare a funcţiei *abs*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    printf("Valoarea absoluta a lui %d e %d\n", 5, abs(5));
    printf("Valoarea absoluta a lui %d e %d\n", 0, abs(0));
    printf("Valoarea absoluta a lui %d e %d\n", -5, abs(-5));
  }
```

Atunci când compilaţi şi executaţi programul *vezi_abs.c*, pe ecran vor fi afişate următoarele:

```
Valoarea absoluta a lui 5 e 5
Valoarea absoluta a lui 0 e 0
Valoarea absoluta a lui -5 e 5
C:\>
```

Observaţie: *Multe compilatoare de C oferă şi funcţia* ***labs****, care returnează valoarea absolută pentru o expresie de tip* ***long int****.*

322 ARCCOSINUS



Arccosinusul este raportul dintre ipotenuza unui triunghi dreptunghic şi latura adiacentă unui unghi ascuţit dat. Cu alte cuvinte, arccosinus este inversul geometric al cosinusului unui unghi. Sau, altfel spus, dacă *y* este cosinusul unui unghi theta, theta este arccosinusul lui *y*. Pentru a ajuta programele dumneavoastră să determine arccosinusul, limbajul C oferă funcţia

acos. Această funcție returnează o valoare de tip *double* ce reprezintă arccosinusul unui unghi în radiani (de la 0 la pi), ca mai jos:

```
#include <math.h>

double acos(double expresie);
```

Dacă expresia dată nu este între –1.0 și 1.0, funcția *acos* va stabili variabila globală *errno* la *EDOM* și va afișa o eroare de tip *DOMAIN* la *stderr*. Următorul program, *veziacos.c*, ilustrează utilizarea funcției *acos*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double radian;

    for (radian = -0.5; radian <= 0.5; radian += 0.2)
      printf("%f %f\n", radian, acos(radian));
  }
```

Observație: *Multe compilatoare de C oferă și funcția* ***acosl****, care returnează arccosinusul unei expresii de tip* ***long double****.*

ARCSINUS

C/C++ 323

Arcsinusul este raportul între ipotenuza unui triunghi dreptunghic și latura opusă unui unghi ascuțit dat. Cu alte cuvinte, arcsinus este inversul geometric al sinusului unui unghi. Sau, altfel spus, dacă *y* este sinusul unui anumit unghi theta, atunci theta este arcsinusul lui *y*. Pentru a ajuta programele dumneavoastră să determine arcsinusul, limbajul C oferă funcția *asin*. Această funcție returnează o valoare de tip *double* ce reprezintă arcsinusul unui unghi în radiani (de la –pi/2 la pi/2), ca mai jos:

```
#include <math.h>

double asin(double expresie);
```

Dacă expresia dată nu este între –1.0 și 1.0, funcția *asin* va stabili variabila globală *errno* la *NAN* și va afișa o eroare de tip *DOMAIN* la *stderr*. Următorul program, *veziasin.c*, ilustrează utilizarea funcției *asin*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double radian;

    for (radian = -0.5; radian <= 0.5; radian += 0.2)
      printf("%f %f\n", radian, asin(radian));
  }
```

Observaţie: *Multe compilatoare de C oferă şi funcţia* ***asinl****, care returnează arcsinusul unei expresii de tip* ***long double****.*

324 ARCTANGENTA



Într-un triunghi dreptunghic, *arctangenta* este raportul între latura adiacentă unui unghi ascuţit dat şi latura opusă lui. Cu alte cuvinte, arctangenta este inversul geometric al tangentei unui unghi. Sau, altfel spus, dacă *y* este tangenta unui anumit unghi theta, atunci theta este arctangenta lui *y*. Pentru a ajuta programele dumneavoastră să determine arctangenta, limbajul C oferă funcţia *atan*. Această funcţie returnează o valoare de tip *double* ce reprezintă arctangenta unui unghi în radiani (de la –pi/2 la pi/2), ca mai jos:

```
#include <math.h>

double atan(double expresie);
```

Următorul program, *veziatan.c*, ilustrează utilizarea funcţiei *atan*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double radian;

    for (radian = -0.5; radian <= 0.5; radian += 0.2)
     printf("%f %f\n", radian, atan(radian));
  }
```

Observaţie: *Multe compilatoare de C oferă şi funcţia* ***atanl****, care returnează arctangenta unei expresii de tip* ***long double****. De asemenea, C oferă funcţiile* ***atan2*** *şi* ***atan2l****, care returnează arctangenta expresiei y/x.*

325 OBŢINEREA VALORII ABSOLUTE A UNUI NUMĂR COMPLEX



Aşa cum aţi învăţat, un număr complex conţine o parte reală şi una imaginară. Funcţiile limbajului C reprezintă numerele complexe ca structuri cu un membru *x* şi un membru *y*, cum se vede mai jos:

```
struct complex
  {
    double x, y;
  };
```

Când lucraţi cu numere complexe, pot apărea situaţii în care trebuie să calculaţi valoarea absolută a numărului (distanţa sa pozitivă faţă de zero). Pentru a permite programului dumneavoastră să calculeze valoarea absolută a unui număr complex, limbajul C oferă funcţia *cabs*, cum se vede mai jos:

```
#include <math.h>

double cabs(struct complex valoare);
```

Funcția *cabs* este similară cu extragerea rădăcinii pătrate din suma pătratelor celor două părți ale numărului complex. În exemplul următor, funcția *cabs* va returna $(10^2 + 5^2)^{1/2}$. Programul *vezicabs.c* ilustrează utilizarea funcției *cabs* în limbajul C:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    struct complex numar_complex;

    numar_complex.x = 10;
    numar_complex.y = 5;
    printf("Valoarea absoluta a lui 10,5 este %f\n",
      cabs(numar_complex));
  }
```

Atunci când compilați și executați programul *vezicabs.c*, pe ecran se va afișa următorul rezultat:

```
Valoarea absoluta a lui 10,5 este 11.180340
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***cabsl****, care returnează valoarea absolută a unui număr complex de tip* ***long double****. Compact-discul care însoțește această carte include un program foarte lung pentru a afla valoarea absolută a unui număr complex, deoarece implementarea numerelor complexe în C++ este mult diferită de cea în C. Programul* ***vezicabs.c*** *se poate compila în ambele medii, C și C++.*

ROTUNJIREA UNEI VALORI REALE ÎN VIRGULĂ MOBILĂ

C/C++ 326

Când lucrați cu valori reale în virgulă mobilă, pot apărea situații în care trebuie să rotunjiți valoarea unei variabile în virgulă mobilă sau a unei expresii, înlocuind-o cu valoarea întreagă imediat următoare. Pentru aceste cazuri, limbajul C oferă funcția *ceil*, cum se vede mai jos:

```
#include <math.h>

double ceil(double valoare);
```

După cum puteți vedea, funcția *ceil* primește un parametru de tip *double* și returnează o valoare de tip *double*. Următorul program, *veziceil.c*, ilustrează modul de utilizare a funcției *ceil*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("Valoarea %f ceil %f\n", 1.9, ceil(1.9));
```

```
    printf("Valoarea %f ceil %f\n", 2.1, ceil(2.1));
  }
```

Atunci când compilați și executați programul *veziceil.c*, pe ecranul dumneavoastră vor fi afișate următoarele:

```
Valoarea 1.900000 ceil 2.000000
Valoarea 2.100000 ceil 3.000000
C:\>
```

***Observație:** Multe compilatoare de C oferă și funcția **ceill**, care rotunjește valori de tipul **long double**.*

327 COSINUSUL UNUI UNGHI



Într-un triunghi, cosinusul unui unghi este raportul între latura adiacentă și ipotenuză. Pentru a permite programelor dumneavoastră să calculeze cosinusul unui unghi, limbajul C oferă funcția *cos*. Această funcție returna o valoare de tip *double* ce reprezintă cosinusul unui unghi specificat în radiani, ca mai jos:

```
#include <math.h>

double cos(double expresie);
```

Funcția *cos* returnează valori în intervalul de la –1.0 la 1.0. Următorul program, *vezi_cos.c*, ilustrează modul de utilizare a funcției *cos*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("Cosinus de pi/2 este %6.4f\n", cos(3.14159/2.0));
    printf("Cosinus de pi este %6.4f\n", cos(3.14159));
  }
```

Atunci când compilați și executați programul *vezi_cos.c*, pe ecranul dumneavoastră vor fi afișate următoarele:

```
Cosinus de pi/2 este 0.0000
Cosinus de pi este -1.0000
C:\>
```

***Observație:** Multe compilatoare de C oferă și funcția **cosl**, care returnează valoarea cosinusului pentru expresii de tip **long double**.*

328 COSINUSUL HIPERBOLIC AL UNUI UNGHI



Cosinusul hiperbolic al unui unghi este cosinusul unui unghi „circular", definit prin intermediul raporturilor de radiani hiperbolici. Pentru a ajuta programele dumneavoastră să determine cosinusul hiperbolic, limbajul C dispune de funcția *cosh*. Această funcție returnează o valoare de tip *double* ce reprezintă cosinusul hiperbolic al unui unghi circular, specificat în radiani, ca mai jos:

```
#include <math.h>
double cosh(double expresie);
```

Dacă apare o depășire, funcția *cosh* returnează valoarea *HUGE_VAL* (sau *_LHUGE_VAL* în cazul funcției *coshl*) și va stabili variabila globală *errno* la *ERANGE*. Următorul program, *vezicosh.c*, ilustrează modul de utilizare a funcției *cosh*:

```
#include <stdio.h>
#include <math.h>
void main(void)
  {
    double radian;
    for (radian = -0.5; radian <= 0.5; radian += 0.2)
      printf("%f %f\n", radian, cosh(radian));
  }
```

Observație: *Multe compilatoare de C oferă și funcția* ***coshl****, care returnează valoarea cosinusului hiperbolic pentru expresii de tip* ***long double****.*

SINUSUL UNUI UNGHI

C/C++329

Într-un triunghi, sinusul unui unghi este raportul între latura opusă și ipotenuză. Pentru a ajuta programele dumneavoastră să determine sinusul unui unghi, limbajul C oferă funcția *sin*. Această funcție returnează o valoare de tip *double* care reprezintă sinusul unui unghi specificat în radiani, ca mai jos:

```
#include <math.h>
double sin(double expresie);
```

Următorul program, *vezi_sin.c*, ilustrează modul de utilizare a funcției *sin*:

```
#include <stdio.h>
#include <math.h>
void main(void)
{
   double radian;
   for (radian = 0.0; radian < 3.1; radian += 0.1)
     printf("Sinus de %f este %f\n", radian, sin(radian));
 }
```

Observație: *Multe compilatoare de C oferă și funcția* ***sinl****, care returnează valoarea sinusului pentru expresii de tip* ***long double****.*

SINUSUL HIPERBOLIC AL UNUI UNGHI

C/C++330

Sinusul hiperbolic al unui unghi este sinusul unui unghi „circular", definit prin intermediul raporturilor de radiani hiperbolici. Pentru a ajuta programele dumneavoastră să determine

sinusul hiperbolic, limbajul C oferă funcția *sinh*. Această funcție returnează o valoare de tip *double* ce reprezintă sinusul hiperbolic al unui unghi circular, specificat în radiani, ca mai jos:

```
#include <math.h>

double sinh(double expresie);
```

Dacă apare o depășire, funcția *sinh* returnează valoarea *HUGE_VAL* (sau *_LHUGE_VAL* în cazul funcției *sinhl*) și va stabili variabila globală *errno* la *ERANGE*. Următorul program, *vezisinh.c*, ilustrează modul de utilizare a funcției *sinh*:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(void)
  {
    double radian;
    double rezultat;

    for (radian = 0.0; radian < 3.1; radian += 0.1)
      if (((rezultat = sinh(radian)) == HUGE_VAL) &&
         (errno == ERANGE))
        printf("Eroare de depasire\n");
      else
        printf("Sinus de %f este \n", radian, rezultat);
  }
```

Observație: *Multe compilatoare de C oferă și funcția* ***sinhl****, care returnează valoarea sinusului hiperbolic pentru expresii de tip* ***long double****.*

331 TANGENTA UNUI UNGHI



Într-un triunghi, tangenta unui unghi este raportul între latura opusă și latura adiacentă. Pentru a ajuta programele dumneavoastră să determine tangenta unui unghi, limbajul C oferă funcția *tan*. Această funcție returnează o valoare de tip *double* care reprezintă tangenta unui unghi specificat în radiani, ca mai jos:

```
#include <math.h>

double tan(double expresie);
```

Următorul program, *vezi_tan.c*, ilustrează modul de utilizare a funcției *tan*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double pi = 3.14159265;

    printf("Tangenta de pi este %f\n", tan(pi));
```

```
    printf("Tangenta de pi/4 este %f\n", tan(pi / 4.0));
  }
```

Când veți compila și executa programul *vezi_tan.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
Tangenta de pi este -0.000000
Tangenta de pi/4 este 1.000000
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția **tanl**, care returnează valoarea tangentei pentru expresii de tip **long double**.*

TANGENTA HIPERBOLICĂ A UNUI UNGHI

C/C++ 332

Tangenta hiperbolică a unui unghi este tangenta unui unghi „circular", definit prin intermediul raporturilor de radiani hiperbolici. Pentru a ajuta programele dumneavoastră să determine tangenta hiperbolică, limbajul C oferă funcția *tanh*. Această funcție returnează o valoare de tip *double* care reprezintă tangenta hiperbolică a unui unghi specificat în radiani, ca mai jos:

```
#include <math.h>

double tanh(double expresie);
```

Observație: *Multe compilatoare de C oferă și funcția **tanhl**, care returnează valoarea tangentei hiperbolice pentru expresii de tip **long double**.*

ÎMPĂRȚIREA NUMERELOR ÎNTREGI

C/C++ 333

Așa cum ați învățat, limbajul C dispune de operatorii diviziune (/) și modulo (%), care permit programelor dumneavoastră să efectueze operații de împărțire sau să determine restul unei operații de împărțire. De asemenea, limbajul C dispune de funcția *div*, care împarte valoarea unui numărător la cea a unui numitor și returnează o structură de tip *div_t*, care conține câtul și restul, ca mai jos:

```
struct div_t
   {
     int cat;
     int rest;
   } div_t;
```

Funcția *div* lucrează cu valori întregi, cum se vede mai jos:

```
#include <math.h>

div_t div(int numarator, int numitor);
```

Următorul program, *divrest.c*, ilustrează modul de utilizare a funcției *div*:

```
#include <stdlib.h>
#include <math.h>
```

```
void main(void)
  {
    div_t rezultat;

    rezultat = div(11, 3);
    printf("11 impartit la 3 este %d Rest %d\n",
    rezultat.cat, rezultat.rest);
  }
```

Atunci când compilați și executați programul *divrest.c*, pe ecranul dumneavoastră se va afișa următoarea ieșire:

```
11 impartit la 3 este 3 Rest 2
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***ldiv****, care returnează câtul și restul pentru valori de tip* ***long****.*

334 LUCRUL CU FUNCȚIA EXPONENȚIALĂ



Atunci când programele dumneavoastră execută operații matematice complexe, pot apărea situații în care este nevoie de calculul exponențial e^x. În asemenea cazuri, programele dumneavoastră pot utiliza funcția *exp*, care returnează o valoare de tip *double*, ca mai jos:

```
#include <math.h>

double exp(double x);
```

Următorul program, *vezi_exp.c*, ilustrează modul în care se utilizează funcția *exp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double val;

    for (val = 0.0; val <= 1.0; val += 0.1)
      printf("exp(%f) este %f\n", val, exp(val));
  }
```

Observație: *Multe compilatoare de C oferă și funcția* ***expl****, care lucrează cu valori de tip* ***long double****.*

335 VALOREA ABSOLUTĂ A EXPRESIILOR ÎN VIRGULĂ MOBILĂ



Așa cum ați învățat, *valoarea absolută* precizează distanța de la valoare la zero. Valoarea absolută este întotdeauna pozitivă. De exemplu, valoarea absolută a lui 2.5 este 2.5. De asemenea, valoarea absolută a lui –2.5 este 2.5. Atunci când lucrați cu valori absolute, pot apărea situații în care trebuie să calculați valoarea absolută a unei expresii în virgulă mobilă.

Pentru asemenea cazuri, limbajul C vă pune la dispoziție funcția *fabs*. Această funcție returnează valoarea absolută a unui număr real în virgulă mobilă, ca mai jos:

```
#include <math.h>

float fabs(float expresie);
```

Următorul program, *vezifabs.c*, ilustrează modul de utilizare a funcției *fabs*:

```
#include <stdio.h>
#include <math.h>

void main(void)
   {
     float val;

     for (val = -1.0; val <= 1.0; val += 0.1)
       printf("Valoarea %f fabs %f\n", val, fabs(val));
   }
```

Observație: *Multe compilatoare de C oferă și funcția* ***fabsl****, care returnează valoarea absolută a unei expresii de tip* ***long double.***

RESTUL ÎN VIRGULĂ MOBILĂ

C/C++ 336

În secțiunea 82, ați învățat cum se folosește operatorul C modulo (%) pentru a obține restul împărțirii unor numere întregi. În funcție de programele dumneavoastră, pot apărea situații în care să aveți nevoie de restul împărțirii unor numere în virgulă mobilă. În aceste cazuri, programele dumneavoastră în C pot utiliza funcția *fmod* pentru a împărți două valori în virgulă mobilă. Funcția *fmod* va returna restul ca valoare în virgulă mobilă, cum se vede mai jos:

```
#include <math.h>

double fmod(double x, double y);
```

De exemplu, dacă invocați *fmod* cu valorile 10.0 și 3.0, funcția va returna valoarea 1.0 (10 împărțit la 3 este 3 rest 1). Următorul program, *vezifmod.c*, ilustrează modul de utilizare a funcției *fmod*:

```
#include <stdio.h>
#include <math.h>

void main(void)
   {
     double numarator = 10.0;
     double numitor = 3.0;
     printf("fmod(10, 3) e %f\n", fmod(numarator, numitor));
   }
```

Atunci când compilați și executați programul *vezifmod.c*, pe ecranul dumneavoastră va fi afișat următorul rezultat:

```
fmod(10, 3) e 1.000000
C:\>
```

Observaţie: *Multe compilatoare de C oferă şi funcţia **fmodl**, care returnează restul fracţionar pentru valorile de tip **long double**.*

337 MANTISA ŞI EXPONENTUL UNEI VALORI ÎN VIRGULĂ MOBILĂ

C/C++

Atunci când programele dumneavoastră lucrează cu valori în virgulă mobilă, calculatorul poate să stocheze valorile utilizând mantisa (a cărei valoare este între 0.5 şi 1.0) şi un exponent, cum se arată în figura 337.

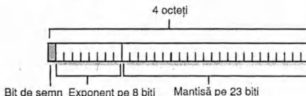


Figura 337 *Calculatorul păstrează valorile în virgulă mobilă utilizând formatul cu mantisă şi exponent.*

Pentru a determina valoarea păstrată, calculatorul combină mantisa şi exponentul, cum se vede mai jos:

```
valoare = mantisa * (2* exponent);
```

De obicei, nu trebuie să vă preocupe utilizarea mantisei şi a exponentului de către calculator. În funcţie de programul dumneavoastră, pot apărea însă situaţii în care trebuie să cunoaşteţi valoarea mantisei şi a exponentului. Pentru asemenea cazuri, limbajul C dispune de funcţia *frexp*, care returnează mantisa şi atribuie exponentul variabilei *exponent*, pe care funcţia apelantă trebuie să o transmită funcţiei *frexp* prin referinţă:

```
#include <math.h>
double frexp(double val, int *exponent);
```

Următorul program, *frexp.c*, ilustrează modul de utilizare a funcţiei *frexp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
   {
    double val = 1.2345;
    double mantisa;
    int exponent;
    mantisa = frexp(val, &exponent);
    printf("Mantisa %f Exponent %d Valoare %f\n", mantisa,
      exponent, mantisa * pow(2.0, 1.0 * exponent));
   }
```

Atunci când compilați și executați programul *frexp.c*, pe ecranul dumneavoastră se va afișa următorul rezultat:

```
Mantisa 0.617250 Exponent 1 Valoare 1.234500
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***frexpl****, care returnează exponentul și mantisa pentru valorile de tip* ***long double****.*

CALCULUL REZULTATULUI OPERAȚIEI X * 2^E C/C++ 338

În secțiunea 334, ați învățat cum se utilizează funcția *exp* în limbajul C pentru a obține rezultatul unei funcții exponențiale e^x. În programele dumneavoastră, puteți să calculați $x*2^e$. În asemenea cazuri, puteți utiliza funcția *ldexp*, ca mai jos:

```
#include <math.h>
double ldexp(double val, int exponent);
```

Următorul program, *ldexp.c*, ilustrează modul de utilizare a funcției *ldexp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("3 *\' 2 la puterea a 4\' este %f\n", ldexp(3.0, 4));
  }
```

Atunci când compilați și executați programul *ldexp.c*, pe ecranul dumneavoastră va fi afișată următoarea ieșire:

```
3 *' 2 la puterea a 4' este 48.000000
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***ldexp****, care acceptă valorile de tip* ***long double****.*

CALCULUL LOGARITMULUI NATURAL C/C++ 339

Logaritmul natural al unui număr este puterea la care trebuie să fie ridicat *e* pentru a se obține numărul respectiv. Pentru a ajuta programele dumneavoastră să determine logaritmul natural, limbajul C dispune de funcția *log*, care returnează logaritmul natural pentru valori reale în virgulă mobilă:

```
#include <math.h>
double log(double val);
```

Dacă valoarea parametrului *val* este mai mică decât 0, funcția *log* va stabili variabila globală *errno* la *ERANGE* și va returna valoarea *HUGE_VAL* (sau *_LHUGE_VAL* în cazul funcției *logl*). Următorul program, *vezi_log.c*, ilustrează modul de utilizare a funcției *log*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("Logaritm natural de 256.0 e %f\n", log(256.0));
  }
```

Atunci când compilaţi şi executaţi programul *vezi_log.c*, pe ecranul dumneavoastră vor fi afişate următoarele:

```
Logaritm natural de 256.0 e 5.545177
C:\>
```

Observaţie: *Multe compilatoare de C oferă şi funcţia* ***logl****, care returnează logaritmul natural pentru valorile de tip* ***long double****.*

340 CALCULUL REZULTATULUI LUI LOG10X



În secţiunea 339, aţi învăţat cum se utilizează funcţia *log* a limbajului C pentru a calcula un logaritm natural. Când veţi scrie programe care efectuează operaţii matematice, puteţi să determinaţi logaritmul în bază 10 al unei valori (notat, de obicei, *log10x*). Pentru aceste cazuri, limbajul C oferă funcţia *log10*, cum se vede mai jos:

```
#include <math.h>

double log10(double val);
```

Dacă valoarea parametrului *val* este 0, funcţia *log10* va stabili variabila globală *errno* la *EDOM* şi va returna valoarea *HUGE_VAL* (sau *_LHUGE_VAL* în cazul funcţiei *log10l*). Următorul program, *log_10.c*, ilustrează modul de utilizare a funcţiei *log_10*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("Logaritmul zecimal al lui 100 este %f\n",
      log10(100.0));
    printf("Logaritmul zecimal al lui 10000 este %f\n",
      log10(10000.0));
  }
```

Atunci când compilaţi şi executaţi programul *log_10.c*, pe ecranul dumneavoastră se vor afişa următoarele:

```
Logaritmul zecimal al lui 100 este 2.000000
Logaritmul zecimal al lui 10000 este 4.000000
C:\>
```

Observaţie: *Multe compilatoare de C oferă şi funcţia* ***log10l****, care acceptă valorile de tip* ***long double****.*

DETERMINAREA VALORILOR MAXIME ȘI MINIME

C/C++ 341

Atunci când programele dumneavoastră compară două numere, puteți să cunoaște valoarea minimă sau maximă dintre două valori. Pentru aceste cazuri, fișierul antet *stdlib.* oferă macroinstrucțiunile *min* și *max*. Următorul program, *min_max.c*, ilustrează modul d utilizare a acestor două macroinstrucțiuni:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    printf("Maximul dintre %f si %f e %f\n", 10.0, 25.0,
           max(10.0, 25.0));
    printf("Minimul dintre %f si %f e %f\n", 10.0, 25.0,
           min(10.0, 25.0));
  }
```

Pentru a înțelege mai bine aceste două macroinstrucțiuni, studiați următoarea implementare

```
#define max(x,y) (((x) > (y)) ? (x) : (y))
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

SEPARAREA UNEI VALORI DOUBLE ÎN COMPONENTELE ÎNTREG ȘI REAL

C/C++ 342

Așa cum ați învățat, o valoare reală în virgulă mobilă este alcătuită din două părți: o part întreagă și o parte fracționară. De exemplu, dacă se consideră numărul 12.345, 12 este parte sa întreagă, iar 0.345 este partea sa fracționară. În programele dumneavoastră, puteți s lucrați cu ambele componente ale valorii – întreagă și fracționară – sau cu fiecare dintre el separat. Pentru aceste cazuri, limbajul C oferă funcția *modf*, cum se vede mai jos:

```
#include <math.h>

double modf(double val, double *parte_intreaga);
```

Funcția *modf* returnează valoarea părții fracționare și atribuie partea întreagă unei anumit variabile. Următorul program, *int_frac.c*, ilustrează modul în care se utilizează funcția *modf*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    double val = 1.2345;
    double parte_intreaga;
    double fract;

    fract = modf(val, &parte_intreaga);
    printf("Valoare %f Parte intreaga %f Parte fractionara %f\n",
```

```
    val, parte_intreaga, fract);
}
```

Atunci când compilați și executați programul *int_frac.c*, pe ecranul dumneavoastră va fi afișată următoarea ieșire:

```
Valoare 1.234500 Parte intreaga 1.000000 Parte fractionara 0.234500
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***modfl****, care returnează partea întreagă și partea fracționară a unei expresii de tip* ***long double****.*

343 CALCULUL REZULTATULUI OPERAȚIEI X^N



Ridicarea unei valori la o putere dată este una dintre operațiile matematice cele mai obișnuite pe care le efectuează programele dumneavoastră. Limbajul C dispune de funcția *pow*, care returnează rezultatul ridicării unei valori la o putere dată, cum se vede mai jos:

```
#include <math.h>

double pow(double val, double putere);
```

Dacă rezultă o depășire din calculul ridicării la puterea dată, funcția *pow* va atribui variabilei globale *errno* valoarea *ERANGE* și va returnează *HUGE_VAL* (sau *_LHUGE_VAL* în cazul funcției *powl*). Următorul program, *vezi_pow.c*, ilustrează modul de utilizare a funcției *pow*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    int putere;
    for (putere = -2; putere <= 2; putere++)
      printf("10 ridicat la %d e %f\n", putere, pow(10.0, putere));
  }
```

Atunci când compilați și executați programul *vezi_pow.c*, pe ecranul dumneavoastră se vor afișa următoarele:

```
10 ridicat la -2 e 0.010000
10 ridicat la -1 e 0.100000
10 ridicat la 0 e 1.000000
10 ridicat la 1 e 10.000000
10 ridicat la 2 e 100.000000
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***powl****, care acceptă valori de tip* ***long double****. De asemenea, fișierul antet* ***complex.h*** *definește un prototip al funcției* ***pow*** *pentru numere complexe.*

CALCULUL REZULTATULUI OPERAȚIEI 10^x

C/C++ 344

În secțiunea 343, ați învățat cum se utilizează funcția *pow* pentru a determina rezultatul unei valori ridicate la o anumită putere. Uneori, programele dumneavoastră vor trebui să calculeze rezultatul operației 10^x. În asemenea cazuri, puteți utiliza funcția *pow* sau, dacă compilatorul dumneavoastră permite (ca în cazul compilatorului *Turbo C++ Lite*), puteți utiliza funcția *pow10*, ca mai jos:

```
#include <math.h>

double pow10(int putere);
```

Următorul program, *pow10.c*, ilustrează modul de utilizare a funcției *pow10*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
    printf("10 ridicat la -1 e %f\n", pow10(-1));
    printf("10 ridicat la 0 e %f\n", pow10(0));
    printf("10 ridicat la 1 e %f\n", pow10(1));
    printf("10 ridicat la 2 e %f\n", pow10(2));
  }
```

Atunci când compilați și executați programul *pow10.c*, pe ecranul dumneavoastră se vor afișa următoarele:

```
10 ridicat la -1 e 0.100000
10 ridicat la 0 e 1.000000
10 ridicat la 1 e 10.000000
10 ridicat la 2 e 100.000000
C:\>
```

Observație: *Multe compilatoare de C oferă și funcția* ***pow10l****, care acceptă valori de tip* ***long double.***

GENERAREA UNUI NUMĂR ALEATOR

C/C++ 345

În programele dumneavoastră, puteți să generați unul sau mai multe numere aleatoare. Pentru asemenea cazuri, limbajul C dispune de două funcții, *rand și random*, care returnează numere întregi, aleatoare, ca mai jos:

```
#include <stdlib.h>

int rand(void);
int random(int limita);
```

Prima funcție, *rand*, returnează un număr întreg, aleator, cuprins în intervalul de la 0 la *RAND_MAX* (valoare definită în *stdlib.h*). Cea de a doua funcție, *random*, returnează un număr întreg, aleator, cuprins într-un interval stabilit de *limită*, unde *limită* este o valoare maximă pe care funcția apelantă o transmite funcției *random*. Următorul program, *random.c*, ilustrează modul de utilizare a ambelor funcții generatoare de numere aleatoare:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    int i;

    printf("Valorile furnizate de rand\n");
    for (i = 0; i < 100; i++)
      printf("%d ", rand());

    printf("Valorile furnizate de random(100))\n");
    for (i = 0; i < 100; i++)
      printf("%d ", random(100));
  }
```

346 PLASAREA VALORILOR ALEATOARE ÎNTR-UN ANUMIT INTERVAL



În secțiunea 345, ați învățat că funcțiile *rand și random* returnează numere aleatoare. Atunci când programele dumneavoastră generează numere aleatoare, pot apărea uneori situații în care trebuie să plaseze aceste valori într-un anumit interval. Dacă lucrați cu valori întregi, puteți folosi funcția *random* și un parametru care specifică valoarea cea mai mare a intervalului numerelor aleatoare. Dacă lucrați însă cu valori în virgulă mobilă, cum ar fi cele din intervalul 0.0 până la 1.0, puteți împărți numărul cu o constantă pentru a obține un număr aleator în virgulă mobilă. Pentru a transforma o serie de numere întregi aleatoare într-o serie de numere în virgulă mobilă, pur și simplu împărțiți numărul aleator la limita superioară a numărului aleator, ca mai jos:

```
random(100)/100.0
```

Exemplul precedent va genera un număr aleator în intervalul 0.01 – 0.99. Dacă programul dumneavoastră necesită un număr aleator în virgulă mobilă cu mai multe cifre, puteți să generați un număr aleator până la 1000 și să-l împărțiți la 1000, ca mai jos:

```
random(1000)/1000.0
```

Exemplul precedent va genera un număr aleator în intervalul 0.001 – 0.999. Dacă programul dumneavoastră necesită numere aleatoare cu o precizie mai mare, pur și simplu măriți numărul aleator întreg maxim și constanta la care veți împărți rezultatul funcției *random*. Următorul program, *plasrand.c*, plasează numere aleatoare în intervalul de la 0.0 până la 1.0 și numere întregi în intervalul de la –5 la 5:

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
  {
    int i;

    printf("Valorile furnizate de random\n");
```

```
for (i = 0; i < 10; i++)
  printf("%f\n", random(100/100.0);
printf("Valorile furnizate de random(-5) pana la random(5)/n");
for (i = 0; i < 100; i++)
  printf("%d\n", random(10)-5);
}
```

LANSAREA GENERATORULUI DE NUMERE ALEATOARE

C/C++ 347

Secțiunea 345 a prezentat funcțiile *rand și random*, pe care le veți utiliza în programele dumneavoastră pentru a genera numerele aleatoare. Când lucrați cu numere aleatoare, pot apărea situații în care trebuie să controlați seria de numere pe care le produce generatorul de numere aleatoare (astfel încât să puteți testa prelucrările programului dumneavoastră cu același set de numere). Alteori, veți dori ca generatorul să creeze aleator numerele. Procesul de atribuire a unui număr de start generatorului de numere aleatoare este denumit *lansare* sau *inițializare* (seeding). Pentru a vă ajuta să lansați generatorul de numere aleatoare, limbajul C prevede două funcții, *randomize* și *srand*, cum se vede mai jos:

```
#include <stdlib.h>

void randomize(void);
void srand(unsigned primul);
```

Prima funcție, *randomize*, utilizează ceasul calculatorului pentru a produce o inițializare aleatoare. Cea de a doua funcție, *srand*, vă permite precizarea valorii de start a generatorului de numere aleatoare. Programele dumneavoastră pot utiliza *srand* pentru a controla intervalul în care vor fi create numerele aleatoare. Următorul program, *randinit.c*, ilustrează modul de utilizare a funcțiilor *srand* și *randomize*:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void main(void)
  {
    int i;

    srand(100);
    printf("Valorile furnizate de rand\n");
    for (i = 0; i < 5; i++)
      printf("%d ", rand());
    printf("\n5 numere identice\n");
    srand(100);
    for (i = 0; i < 5; i++)
      printf("%d ", rand());
    randomize();
    printf("\n5 numere diferite\n");
    for (i = 0; i < 5; i++)
```

```
    printf("%d ", rand());
  }
```

348 CALCULUL RĂDĂCINII PĂTRATE A UNEI VALORI C/C++

Atunci când programele dumneavoastră calculează expresii matematice, deseori trebuie să efectuați operația de extragere a rădăcinii pătrate. Pentru a ajuta programele dumneavoastră să efectueze operația de extragere a rădăcinii pătrate, limbajul C oferă funcția *sqrt*, cum se vede mai jos:

```
#include <math.h>
double sqrt(double val);
```

Funcția *sqrt* lucrează numai cu valori pozitive. Dacă programul dumneavoastră invocă funcția *sqrt* cu valori negative, ea stabilește variabila globală *errno* la *EDOM*. Următorul program, *sqrt.c*, ilustrează modul de utilizare a funcției *sqrt*:

```
#include <stdio.h>
#include <math.h>

void main(void)
  {
   double val;
   for (val = 0.0; val < 10.0; val += 0.1)
     printf("Valoarea %f radacina patrata %f\n", val, sqrt(val));
  }
```

Observație: *Multe compilatoare de C oferă și funcția* ***sqrtl****, care returnează rădăcina pătrată a unei valori de tip* ***long double****.*

349 TRATAREA PERSONALIZATĂ A ERORILOR MATEMATICE C/C++

Câteva dintre funcțiile prezentate în acest capitol detectează erorile de interval și de depășire. În mod prestabilit, când apar astfel de erori, funcțiile apelează o funcție specială, numită *matherr*, care execută unele procese suplimentare, cum ar fi atribuirea unui anumit număr de eroare variabilei globale *errno*. Când apare o asemenea situație, dacă programele dumneavoastră își definesc propria lor funcție *matherr*, rutinele matematice ale limbajului C vor invoca programul dumneavoastră de tratare a erorilor. Atunci când rutinele matematice invocă funcția dumneavoastră *matherr*, ele vor transmite acesteia un pointer la o variabilă de tip *exception*, ca mai jos:

```
struct exception
  {
   int type;
   char *name;
   double arg1, arg2, retval;
  };
```

Membrul *type* conține o constantă care descrie tipul erorii. Tabelul 349 descrie valorile de eroare.

Valoare de eroare	Semnificație
DOMAIN	Un argument nu este în domeniul de valori acceptat de funcție
OVERFLOW	Un argument produce un rezultat care depășește tipul rezultat
SING	Un argument produce un rezultat de singularitate
TLOSS	Un argument produce un rezultat în care toate cifrele de precizie s-au pierdut
UNDERFLOW	Un argument produce un rezultat care depășește inferior tipul rezultat

Tabelul 349 *Constantele limbajului C care descriu erorile matematice.*

Membrul *name* conține numele rutinei care a întâlnit eroarea. Membrii *arg1* și *arg2* conțin parametrii pe care funcția care a întâlnit eroarea îi transmite către *matherr*, în timp ce *retval* conține o valoare de revenire prestabilită (pe care o puteți atribui). Dacă *matherr* nu poate determina cauza precisă a erorii, ea va afișa pe ecran un mesaj generic de eroare. Următorul program, *matherr.c*, ilustrează modul de tratare personalizată a erorilor:

```
#include <stdio.h>
#include <math.h>
void main(void)
  {
    printf("Radacina patrata a lui -1 este %f\n", sqrt(-1.0));
  }
int matherr(struct exception *eroare)
  {
    switch (eroare->type)
      {
        case DOMAIN:      printf("Eroare de domeniu\n");
                          break;
        case PLOSS:       printf("Eroare de pierdere partiala a
                            preciziei\n");
                          break;
        case OVERFLOW:    printf("Eroare de depasire\n");
                          break;
        case SING:        printf("Eroare de singularitate\n");
                          break;
        case TLOSS:       printf("Eroare de pierdere totala a
                            preciziei\n");
                          break;
        case UNDERFLOW:   printf("Eroare de depasire
                           inferioara\n");
                          break;
      };
    printf("Eroarea a aparut in %s valoare %f\n", eroare->name,
      eroare->arg1);
```

```
    eroare->retval = 1;
    return(1);
  }
```

Observaţie: *Funcţia **matherr** poate găsi numai erorile de domeniu şi de depăşire. Pentru a detecta erorile de împărţire la zero, folosiţi **signal**. Multe compilatoare de C oferă suport şi pentru funcţia **matherrl**, care acceptă argumente de tip **long double**.*

350 DETERMINAREA UNITĂŢII CURENTE DE DISC



Dacă programele dumneavoastră lucrează în mediul DOS, pot apărea situaţii în care trebuie să determinaţi unitatea curentă de disc. În aceste cazuri, programele pot utiliza funcţia *getdisk*, cum se vede mai jos:

```
#include <dir.h>

int getdisk(void);
```

Funcţia returnează numărul unităţii, unde 1 este unitatea A, 2 este unitatea B şi aşa mai departe. Următorul program, *da_unit.c*, ilustrează modul de utilizare a funcţiei *getdisk* pentru afişarea literei unităţii curente de disc:

```
#include <stdio.h>
#include <dir.h>

void main(void)
  {
    printf("Unitatea curenta este %c\n", getdisk() + 'A');
  }
```

Observaţie: *Compact-discul care însoţeşte această carte conţine fişierul **win_getd.cpp**, care execută aceeaşi funcţie ca şi programul **da_unit.c**, dar lucrează numai sub Windows 95 sau Windows NT.*

351 SELECTAREA UNITĂŢII CURENTE



În secţiunea 350, aţi învăţat cum se foloseşte funcţia *getdisk* pentru a determina unitatea de disc curentă în mediul DOS. Aşa cum uneori programele dumneavoastră trebuie să determine unitatea curentă, alteori ele trebuie să selecteze o anumită unitate. În asemenea cazuri, programele dumneavoastră pot utiliza funcţia *setdisk*, cum se arată în continuare:

```
#include <dir.h>

int setdisk(int unitate);
```

Parametrul *unitate* este o valoare de tip întreg care precizează unitatea dorită, unde 0 este unitatea A, 1 este unitatea B şi aşa mai departe. Funcţia returnează numărul unităţilor de disc prezente în sistem. Următorul program, *select_c.c*, foloseşte funcţia *setdisk* pentru a selecta unitatea C ca unitate curentă. Programul afişează, de asemenea, numărul total de unităţi disponibile (aşa cum stabileşte intrarea LASTDRIVE din fişierul *config.sys*):

```
#include <stdio.h>
#include <dir.h>

void main(void)
  {
    int nr_unitate;
    nr_unitate = setdisk(3);
    printf("Numarul de unitati disponibile este %d\n", nr_unitate);
  }
```

Observație: *Compact-discul care însoțește această carte conține fișierul* ***win_setd.cpp****, care execută aceeași funcție ca și programul* ***select_c.c****, dar lucrează numai sub Windows 95 sau Windows NT.*

DETERMINAREA SPAȚIULUI DISPONIBIL PE DISC

C/C++ 352

Atunci când programele dumneavoastră păstrează o cantitate însemnată de informații pe un disc – indiferent dacă este vorba de dischetă, hard-disc sau alt tip – fiecare program trebuie să cunoască cât spațiu este disponibil pe disc, pentru a reduce posibilitatea depășirii spațiului pe durata unei operații critice cu discul. Dacă lucrați într-un sistem bazat pe mediul DOS, programele dumneavoastră pot utiliza funcția *getdfree*. Funcția *getdfree* returnează o structură de tip *dfree*, cum se arată în continuare:

```
struct dfree
  {
    unsigned df_avail;       //Clustere disponibile
    unsigned df_total;       // Total clustere
    unsigned df_bsec;        // Octeti pe sector
    unsigned df_sclus;       // Sectoare pe cluster
  };
```

Formatul funcției *getdfree* este următorul:

```
#include <dos.h>

void getdfree(unsigned char unitate, struct dfree *dtable);
```

Parametrul *unitate* precizează unitatea dorită, unde 1 este unitatea A, 2 este unitatea B și așa mai departe. Următorul program, *diskfree.c*, utilizează funcția *getdfree* pentru a obține informații referitoare la unitatea de disc curentă:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct dfree diskinfo;
    long disc_spatiu;

    getdfree(3, &diskinfo);
```

```
    disc_spatiu =     (long) diskinfo.df_avail *
                      (long) diskinfo.df_bsec *
                      (long) diskinfo.df_sclus;
    printf("Spatiul disponibil pe disc %ld\n", disc_spatiu);
}
```

Observaţie: *Compact-discul care însoţeşte această carte conţine fişierul* ***win_free.cpp****, care execută aceeaşi funcţie ca şi programul* ***diskfree.c****, dar lucrează numai sub Windows 95 sau Windows NT.*

353 TESTAREA COMPRIMĂRII CU DBLSPACE

C/C++

Câteva secţiuni din acest capitol v-au arătat modalitatea de executare a operaţiilor de citire şi scriere absolută, care lucrează cu sectoarele discului. Înainte ca programele dumneavoastră să efectueze operaţii de I/O de nivel jos, asiguraţi-vă că discul pe care îl veţi citi nu este comprimat cu *dblspace* sau alt utilitar. Discurile comprimate păstrează informaţia pe baza principiului sector-după-sector. Dacă scrieţi pe un sector de disc comprimat, vă asumaţi un considerabil risc de alterare a discului, pierzând informaţia conţinută. De regulă, cele mai multe programe nu necesită efectuarea unor asemenea operaţii de nivel jos de citire şi scriere. Dacă scrieţi un program utilitar pentru disc, cum ar fi *undelete*, trebuie să ştiţi cum se testează faptul că un disc este comprimat şi cum se lucrează cu acesta.

354 CITIREA INFORMAŢIILOR DIN FAT

C/C++

Dacă lucraţi într-un sistem bazat pe mediul DOS, tabela de alocare a fişierelor (*FAT*) urmăreşte care părţi ale discului sunt utilizate, care părţi sunt deteriorate şi care sunt disponibile (pentru păstrarea fişierelor şi a programelor). Dacă programele dumneavoastră efectuează operaţii de disc de nivel jos, uneori veţi avea nevoie de informaţii cum ar fi tipul discului, numărul de octeţi pe sector, numărul de sectoare pe cluster sau numărul de clustere pe disc. În asemenea cazuri, programele dumneavoastră pot utiliza funcţiile *getfat* şi *getfatd*, ca mai jos:

```
#include <dos.h>

void getfat(unsigned char unitate, struct fatinfo *fat);
void getfatd(struct fatinfo *fat);
```

Funcţia *getfat* vă permite să precizaţi unitatea dorită, pe când funcţia *getfatd* returnează informaţii pentru unitatea curentă. Pentru a preciza funcţiei *getfat* litera unităţii, introduceţi valoarea 1 pentru unitatea A, 2 pentru B, 3 pentru C şi aşa mai departe. Funcţiile *getfat* şi *getfatd* atribuie informaţia unei structuri de tip *fatinfo*, ca mai jos:

```
struct fatinfo
  {
    char fi_sclus;          // Sectoare pe cluster
    char fi_fatid;          // Tip disc
    unsigned fi_nclus;      // Clustere pe disc
    int fi_bysec;           // Octeti pe sector
  };
```

Următorul program, *getfatd.c*, utilizează funcția *getfatd* pentru afișarea informațiilor despre unitatea de disc curentă:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct fatinfo fat;
    getfatd(&fat);
    printf("Sectoare pe cluster %d\n", fat.fi_sclus);
    printf("Clustere pe disc %u\n", fat.fi_nclus);
    printf("Octeti pe sector %d\n", fat.fi_bysec);
    printf("Tip disc %x\n", fat.fi_fatid & 0xFF);
  }
```

Observație: *Dacă aveți instalat pe calculatorul dumneavoastră Windows NT și aveți unitatea partiționată ca unitate NTFS (NT File System), nu există o tabelă FAT pe care să o puteți accesa. Pentru a afla mai mult despre NTFS, vizitați situl Web al Departamentului de Informatică al Universității Yale, la adresa* ***http://pclt.cis.yale.edu/pclt/BOOT/IFS.HTM.***

IDENTIFICATORUL DISCULUI

C/C++ 355

În secțiunea 354, ați folosit funcțiile *getfat* și *getfatd* pentru a afla informații despre unitatea de disc curentă. Cum ați văzut, aceste funcții întorc un octet numit *fi_fatid*, care conține identificatorul discului DOS. Tabelul 355 prezintă valorile pe care le poate lua *fi_fatid*:

Valoare (hex)	Tip disc
F0H	3 1/2 inch 1,44 MB sau 2.88 MB
	Disc Zip
F8H	Hard-disc
	CD-ROM
F9H	3 1/2 inch 720 KB sau 5 1/4 inch 1,2 MB
FAH	5 1/4 inch 320 KB
FCH	5 1/4 inch 180 KB
FDH	5 1/4 inch 360 KB
FEH	5 1/4 inch 160 KB
FFH	5 1/4 inch 320 KB

Tabelul 355 *Valorile identificatorului de disc returnat de DOS.*

Observație: *Compact-discul care însoțește această carte conține fișierul* ***win_did.cpp****, care listează valorile identificatorului de disc sub Windows 95 sau Windows NT și le afișează pe ecran.*

356 EXECUTAREA UNEI CITIRI/SCRIERI ABSOLUTE DE SECTOR

C/C++

Dacă lucraţi în mediul DOS, puteţi să executaţi operaţii de citire şi scriere absolută, la nivel de sector. În mod normal, programele dumneavoastră execută aceste operaţii cu ajutorul serviciilor DOS. Pentru a simplifica executarea lor, multe compilatoare de C oferă funcţiile *absread* şi *abswrite*, ca mai jos:

```
#include <dos.h>

void absread(int unitate, int nr_sect, long sect_start,
  void *buffer);
void abswrite(int unitate, int nr_sect, long sect_start, void
  *buffer);
```

Parametrul *unitate* precizează unitatea de disc pe care doriţi s-o citiţi, unde 0 este A, 1 este B şi aşa mai departe. Parametrul *nr_sect* specifică numărul de sectoare pe care vreţi să le citiţi sau să le scrieţi, începând cu sectorul precizat de parametrul *sect_start*. În sfârşit, parametrul *buffer* este un pointer la bufferul unde este citită informaţia sau unde este scrisă ieşirea. Dacă funcţiile se execută cu succes, ele returnează valoarea 0. Dacă apare o eroare, ele returnează valoarea –1. Următorul program, *chk_disk.c*, citeşte fiecare sector al unităţii C. Dacă programul găseşte o eroare la citirea sectorului, el va afişa numărul acelui sector:

```
#include <stdio.h>
#include <dos.h>
#include <alloc.h>

void main(void)
  {
    struct fatinfo fat;
    long sector, total_sect;
    void *buffer;

    getfat(3, &fat);
    total_sect = fat.fi_nclus * fat.fi_sclus;
    if ((buffer = malloc(fat.fi_bysec)) == NULL)
      printf("Eroare la alocarea bufferului\n");
    else
      for (sector = 0; sector < total_sectors; sector++)
        if (absread(2, 1, sector, buffer) == -1)
          {
            printf("\n\007Eroare la citirea sectorului %ld apasati
              tasta Enter\n", sector);
            getchar();
          }
        else
          printf("Se citeste sectorul %ld\r", sector);
  }
```

Observaţie: *Puteţi executa citiri sau scrieri absolute de sector în Windows, dar modul în care Windows scrie informaţiile pe disc face ca operaţiile de citire şi scriere absolută să fie*

periculoase și incoerente. Trebuie să evitați activitățile absolute cu discul în Windows și să executați operațiile de citire și scriere prin API.

EXECUTAREA OPERAȚIILOR I/O CU DISCUL PRIN BIOS

C/C++357

Atunci când programele dumneavoastră execută operații cu fișiere, ele utilizează serviciile DOS de manipulare a fișierelor. Aceste servicii, la rândul lor, apelează alte servicii DOS pentru citirea și scrierea sectoarelor logice ale discului. Pentru efectuarea propriu-zisă a operațiilor I/O cu discul, serviciile DOS apelează serviciile de disc BIOS. De exemplu, dacă scrieți programe utilitare pentru disc, s-ar putea ca acestea să aibă nevoie de operații I/O cu discul de nivel jos. În asemenea cazuri, programele dumneavoastră pot utiliza funcția *biosdisk*, cum se vede mai jos:

```
#include <bios.h>

int biodisk(int operatie, int unitate, int head, int track,
  int sector, int nr_ sector, void *buffer);
```

Parametrul *unitate* precizează numărul unității, care este 0 pentru A, 1 pentru B și așa mai departe. Pentru hard-discuri, 0x80 este prima unitate, 0x81 a doua și așa mai departe. Parametrii *head, track, sector* și *nr_sector* precizează sectoarele fizice ale discului pe care vreți să scrieți sau să citiți. Parametrul *buffer* este un pointer la bufferul unde funcția *biosdisk* citește date sau scrie date. În sfârșit, parametrul *operatie* specifică funcția dorită. Tabelul 357.1 prezintă operațiile valide.

Operație	Funcție
0	Inițializează sistemul de disc
1	Returnează starea ultimei operații pe disc
2	Citește numărul precizat de sectoare
3	Scrie numărul precizat de sectoare
4	Verifică numărul precizat de sectoare
5	Formatează pista precizată – bufferul conține un tabel cu locațiile defecte
6	Formatează pista precizată și marchează sectoarele defecte
7	Formatează unitatea începând cu pista specificată
8	Returnează parametrii unității de disc în primii patru octeți ai bufferului
9	Inițializează unitatea de disc
10	Execută o citire lungă – 512 octeți de sector plus patru suplimentari
11	Execută o scriere lungă – 512 octeți de sector plus patru suplimentari
12	Execută o poziționare pe disc
13	Inițializarea alternativă a discului
14	Citește bufferul sectorului
15	Scrie bufferul sectorului
16	Testează dacă unitatea este pregătită

(continuare)

Operație	Funcție
17	Recalibrează unitatea
18	Execută diagnosticarea controlerului de RAM
19	Execută diagnosticarea unității
20	Execută diagnosticarea internă a controlerului

Tabelul 357.1 *Operațiile permise de funcția* ***biosdisk***

Dacă se execută cu succes, funcția returnează valoarea 0. Dacă apare o eroare, valoarea returnată de funcție precizează eroarea. Tabelul 357.2 prezintă valorile de eroare.

Valoare	Eroare
0	Succes
1	Comandă nevalabiă
2	Adresa nu a fost găsită
3	Disc protejat la scriere
4	Sectorul nu a fost găsit
5	Inițializarea hard-discului eșuată
6	Linia de schimbare a discului
7	Activitate eșuată a parametrului unitate
8	Depășire DMA
9	DMA dincolo de limita de 64 KB
10	Sector defect
11	Pistă defectă
12	Pistă inexistentă
16	Eroare CRC/ECC la citire
17	Date corectate cu CRC/ECC
32	Eroare de controler
64	Poziționare eșuată
128	Lipsă răspuns
170	Hard-discul nu e pregătit
187	Eroare nedefinită
204	Eroare de scriere
224	Eroare de stare
255	Sesizare eșuată

Tabelul 357.2 *Valorile de eroare pe care le returnează funcția* ***biosdisk.***

Observație: *Multe compilatoare oferă și o funcție numită* ***_bios_disk***, *care execută aceleași procese ca și funcția* ***biosdisk***, *cu excepția faptului că programele dumneavoastră transmit funcției o structură de tip* ***diskinfo_t***, *care conține valorile:* ***drive, head, track, sector*** *și* ***sector_count.***

Observație: *Puteți utiliza funcția* ***bios_disk*** *pentru a executa operații I/O cu discul prin BIOS sub Windows, dar, datorită metodelor pe care Windows le utilizează pentru a scrie informațiile pe disc, aceste operații sunt periculoase și incoerente. În Windows, trebuie să evitați operațiile I/O cu discul prin BIOS și să executați citirea/scrierea pe disc prin funcțiile API.*

TESTAREA ACCESIBILITĂȚII UNITĂȚII DE DISCHETE

C/C++ 358

În secțiunea 357, ați învățat cum se folosește funcția *biosdisk* pentru a invoca serviciile BIOS pentru disc. O operație utilă pe care funcția *biosdisk* poate să o execute, este să testeze dacă unitatea de dischete conține un disc și este gata pentru acces. Următorul program, *test_a.c*, utilizează funcția *biosdisk* pentru verificarea unității de dischete:

```
#include <stdio.h>
#include <bios.h>

void main(void)
  {
    char buffer[8192];

    // Citirea zonei cap 1, pista 1, sector 1
    if (biosdisk(2, 0, 1, 1, 1, 1, buffer))
      printf("Eroare de accesare a unitatii\n");
    else
      printf("Unitate pregatita\n");
  }
```

Observație: *Compact-discul însoțitor al acestei cărți include fișierul* ***win_a.cpp****, care execută aceeași funcție ca și programul* ***test_a.c****, dar lucrează numai sub Windows 95 sau Windows NT.*

DESCHIDEREA UNUI FIȘIER CU FOPEN

C/C++ 359

Multe programe în C create de dumneavoastră păstrează și regăsesc informația în fișiere. Înainte de a putea citi sau scrie informația în fișier, programul trebuie să deschidă fișierul. Funcția *fopen* permite programelor dumneavoastră să deschidă un fișier. Formatul funcției *fopen* este următorul:

```
#include <stdio.h>

FILE *fopen(const char *numefisier, const char *mod);
```

Parametrul *numefisier* este un șir de caractere care conține numele fișierului dorit, cum ar fi: *„c:\datafile.dat"*. Parametrul *mod* precizează cum vreți să utilizați fișierul – pentru citire, scriere sau adăugare. Tabelul 359 descrie valorile modurilor acceptate de funcția *fopen*.

Mod	Semnificație
a	Deschide fișierul pentru operații de adăugare – dacă fișierul nu există, sistemul de operare îl creează
r	Deschide un fișier existent pentru operații de citire
w	Deschide un fișier nou pentru ieșire – dacă există un fișier cu același nume, sistemul de operare îl suprascrie
r+	Deschide un fișier existent pentru citire și scriere
w+	Deschide un fișier nou pentru citire și scriere – dacă există un fișier cu același nume, sistemul de operare îl suprascrie
a+	Deschide un fișier pentru operații de adăugare și citire – dacă fișierul nu există, sistemul de operare îl creează

Tabelul 359 *Modurile acceptate de funcția* ***fopen.***

Funcția *fopen* returnează un pointer (numit *pointer de fișier*) la o structură de tip *FILE* pe care o definește fișierul antet *stdio.h*. Programele dumneavoastră folosesc pointerul de fișier pentru operațiile de intrare/ieșire. Dacă funcția *fopen* nu poate deschide un anumit fișier, ea returnează valoarea *NULL*. Programele dumneavoastră trebuie să testeze întotdeauna valoarea returnată de funcția *fopen* pentru a se asigura deschiderea cu succes a fișierului, cum se vede mai jos:

```
if ((pointer_fisier = fopen("FILENAME.EXT", "r")) != NULL)
   {
     // A reusit deschiderea fisierului
   }
else
   {
     // Eroare la deschiderea fisierului
   }
```

În cadrul programelor dumneavoastră, trebuie să declarați variabila pointer de fișier, ca mai jos:

```
void main(void)
   {
     FILE *pointer_fisier; // Pointer la o structura de tip FILE
```

Multe programe deschid un fișier pentru intrare și altul pentru ieșire. În asemenea cazuri, veți declara doi pointeri de fișier, cum se vede mai jos:

```
FILE *intrare, *iesire;
```

Multe secțiuni din acest capitol folosesc funcția *fopen* pentru a deschide fișiere pentru operații de citire, scriere sau adăugare.

360 STRUCTURA FILE



Așa cum ați învățat, atunci când programele dumneavoastră execută operații de intrare/ieșire cu fișierele, declară în mod obișnuit pointeri de fișier utilizând structura *FILE*, cum se vede mai jos:

```
FILE *intrare, *iesire;
```

Dacă examinați fișierul antet *stdio.h*, veți găsi definiția structurii *FILE*. În cazul compilatorului *Turbo C++ Lite*, structura are forma următoare:

```
typedef struct
  {
    short level;                // Nivel plin/gol al bufferului
    unsigned flags;             // Indicatoare de stare fisier
    char fd;                    // Descriptor fisier
    unsigned char hold;         // ungetc char daca nu e buffer
    short bsize;                // Dimensiune buffer
    unsigned char *buffer;      // Buffer transfer date
    unsigned char *curp;        // Pointerul activ curent
    unsigned istemp;            // Indicator de fisier temporar
    short token;                // Folosit pentru testare
                                // validitate
  } FILE;                       // Acesta este obiectul FILE
```

Structura *FILE* conține *descriptorul de fișier* de nivel jos pe care sistemul de operare îi folosește pentru a accesa fișierele, dimensiunea bufferului fișierului și locația sa, bufferul de caractere pe care îl folosește *unget*, un indicator care arată dacă fișierul este temporar și alte variabile de tip indicator (flag). În plus, structura *FILE* păstrează pointerul de fișier care ține evidența locației curente în interiorul fișierului.

Dacă lucrați într-un mediu DOS, cele mai multe compilatoare definesc un tablou cu dimensiune fixă (de obicei 20) de pointeri de fișier care găzduiesc informația pentru fiecare fișier pe care îl deschide programul dumneavoastră. Dacă programul trebuie să deschidă mai mult de 20 de fișiere, trebuie să studiați documentația care însoțește compilatorul pentru a vă informa ce pași trebuie făcuți în vederea modificării dimensiunii tabloului de pointeri de fișier.

ÎNCHIDEREA UNUI FIȘIER DESCHIS

C/C++ 361

Așa cum programele dumneavoastră trebuie să deschidă fișierele înainte de a le folosi, trebuie, de asemenea, să închidă fișierele de care nu mai au nevoie. Închiderea unui fișier cere sistemului de operare să golească toate bufferele discului asociate fișierului și să elibereze resursele de sistem consumate de fișier, cum ar fi datele pointerului de fișier. Funcția *fclose* a limbajului C închide fișierul asociat pointerului de fișier specificat, ca mai jos:

```
#include <stdio.h>

int fclose(FILE *pointer_fisier);
```

Dacă funcția se execută cu succes, returnează valoarea 0. Dacă apare o eroare, *fclose* returnează constanta *EOF*, cum se vede mai jos:

```
if (fclose(pointer_fisier) == EOF)
  printf("Eroare la inchiderea fisierului\n");
```

Dacă veți studia diferite programe în C, veți observa că majoritatea nu testează valoarea returnată de funcția *fopen*, ca mai jos:

```
fclose(pointer_fisier);
```

În cele mai multe cazuri, dacă la închiderea unui fișier apare o eroare, programul nu poate să facă prea multe pentru a corecta situația. Dacă însă lucrați cu fișiere care conțin date critice, trebuie să afișați un mesaj de eroare către utilizator, pentru ca acesta să examineze conținutul fișierului.

***Observație:** Dacă nu invocați funcția **fclose**, compilatorul de C va închide la sfârșitul programului fișierele pe care le-ați deschis.*

362 CITIREA ȘI SCRIEREA INFORMAȚIILOR ÎN FIȘIER CARACTER CU CARACTER



Atunci când programele dumneavoastră execută operații I/O cu fișiere, ele pot citi sau scrie fie câte un caracter, fie câte o linie. Pentru operațiile de intrare și ieșire cu caractere, programele dumneavoastră pot utiliza funcțiile *fgetc* și *fputc*, al căror format este cel de mai jos:

```
#include <stdio.h>

int fgetc(FILE *pointer_intrare);
int fputc(int caracter, FILE *pointer_iesire);
```

Funcția *fgetc* citește caracterul curent dintr-un anumit fișier de intrare. Dacă pointerul de fișier a ajuns la sfârșitul fișierului, funcția *fgetc* returnează constanta *EOF*. Funcția *fputc* scrie un caracter în locația curentă a fișierului de ieșire specificat. Dacă apare o eroare, funcția *fputc* returnează constanta *EOF*. Următorul program, *confcopi.c*, utilizează funcțiile *fgetc* și *fputc* pentru a copia conținutul fișierului *config.sys* din directorul rădăcină într-un fișier numit *config.tst*:

```
#include <stdio.h>

void main(void)
  {
   FILE *intrare, *iesire;
   int litera;

   if ((intrare = fopen("\\CONFIG.SYS", "r")) == NULL)
     printf("Eroare la deschiderea \\CONFIG.SYS\n");
   else if ((iesire = fopen("\\CONFIG.TST", "w")) == NULL)
     printf("Eroare la deschiderea \\CONFIG.TST\n");
   else
     {
       // Scrie si citeste fiecare caracter din fisier
       while ((litera = fgetc(intrare)) != EOF)
         fputc(litera, iesire);
       fclose(intrare); // Inchide fisier intrare
       fclose(iesire); // Inchide fisier iesire
     }
  }
```

POINTERUL DE POZIȚIE AL UNUI FIȘIER

C/C++ 363

Secțiunea 360 a prezentat structura *FILE*. Așa cum ați învățat, unul dintre câmpurile structurii conține un *pointer de poziție*, care indică locația curentă în interiorul fișierului. Când deschideți pentru prima oară un fișier pentru operații de citire sau scriere, sistemul de operare stabilește pointerul de poziție la începutul fișierului. De fiecare dată când citiți sau scrieți un caracter, pointerul de poziție avansează cu un caracter. Dacă citiți o linie de text, pointerul de poziție avansează la începutul liniei următoare. Folosind pointerul de poziție, funcțiile de intrare sau de ieșire pot să țină evidența locației curente în cadrul fișierului. Când deschideți un fișier pentru adăugare, sistemul de operare stabilește pointerul de poziție la sfârșitul fișierului. În următoarele capitole, veți învăța cum se fixează pointerul de poziție la o anumită locație de fișier folosind funcțiile *fseek* și *fsetpos*. Tabelul 363 precizează locația la care funcția *fopen* plasează pointerul de poziție când deschideți fișierul pentru citire, scriere sau adăugare.

Mod de deschidere	Poziție în fișier
a	Imediat după ultimul caracter din fișier
r	La începutul fișierului
w	La începutul fișierului

Tabelul 363 *Poziția fixată în fișier ca rezultat al apelării funcției* ***fopen.***

DETERMINAREA POZIȚIEI CURENTE A FIȘIERULUI

C/C++ 364

În secțiunea 363, ați învățat cum urmărește compilatorul de C poziția curentă în fișierele deschise pentru operații de intrare sau ieșire. În funcție de programele dumneavoastră, poate că veți dori să determinați uneori valoarea pointerului de poziție. În aceste cazuri, programele dumneavoastră pot utiliza funcția *ftell*, ca mai jos:

```
#include <stdio.h>
long int ftell(FILE *pointer_fisier);
```

Funcția *ftell* returnează o valoare de tip *long int* care precizează deplasamentul octetului din poziția curentă a unui anumit fișier. Următorul program, *vezi_poz.c*, utilizează funcția *ftell* pentru a afișa datele pointerului de poziție. Programul începe cu deschiderea în modul de citire a fișierului *config.sys* din directorul rădăcină. Programul utilizează apoi funcția *ftell* pentru a afișa poziția curentă. În continuare, programul citește și afișează conținutul fișierului. După ce a găsit sfârșitul fișierului, programul folosește din nou funcția *ftell* pentru a afișa poziția curentă, cum se vede mai jos:

```
#include <stdio.h>
void main(void)
  {
    FILE *intrare;
    int litera;

    if ((intrare = fopen("\\CONFIG.SYS", "r")) == NULL)
      printf("Eroare la deschiderea \\CONFIG.SYS\n");
```

```
    else
      {
        printf("Pozitia curenta este octetul %d\n\n",
         ftell(intrare));
          // Citeste si scrie fiecare caracter din fisier
        while ((litera = fgetc(intrare)) != EOF)
          fputc(litera, stdout);
        printf("\nPozitia curenta este octetul %d\n",
         ftell(intrare));
        fclose(intrare); // Inchide fisierul de intrare
      }
  }
```

365 FLUXURILE



Multe cărți și reviste consideră pointerii de fișier din limbajul C ca *pointeri de fluxuri*. Spre deosebire de alte limbaje de programare, C nu presupune că fișierele conțin informații într-un anumit format. Limbajul C consideră că orice fișier este pur și simplu o colecție de octeți. Atunci când citiți un fișier, îl parcurgeți octet după octet, ca pe un flux de octeți. Interpretarea octeților este lăsată pe seama programelor sau funcțiilor dumneavoastră. De exemplu, funcția *fgets* consideră caracterul *avans de rând* ca sfârșit al unei linii și început al alteia. Funcția *fgets* interpretează acest caracter în mod automat. Cu alte cuvinte, nu limbajul C interpretează octeții. Atunci când scrieți programe și funcții care lucrează cu fișiere, gândiți-vă la fișiere ca la o colecție de octeți și nimic mai mult.

366 CONVERSIA FIȘIERELOR



Funcțiile C de manipulare a fișierelor, cum ar fi *fgets* și *fputs*, pot interpreta fișierele în două moduri: *text* sau *binar*. Modul prestabilit al funcțiilor *fgets* și *fputs* este cel de tip text. În acest mod, funcțiile din categoria *fputs*, care scriu informații într-un fișier, convertesc caracterul *avans de rând* în combinația *retur de car* cu *avans de rând*. În timpul unei operații de intrare, funcțiile din categoria *fgets*, convertesc combinația *retur de car* cu *avans de rând* într-un singur caracter *avans de rând*. În modul de tip binar, pe de altă parte, funcțiile nu execută aceste conversii de caractere. Pentru a vă ajuta să determinați modul curent de interpretare, multe compilatoare din mediul DOS și Windows dispun de variabila globală *_fmode*, care conține una dintre valorile prezentate în tabelul 366.

Constantă	Descriere
O_TEXT	Interpretare de tip text
O_BINARY	Interpretare de tip binar

***Tabelul 366** Constantele atribuite variabilei **_fmode**.*

În mod prestabilit, valoarea variabilei *_fmode* atât sub DOS, cât și sub Windows, este *O_TEXT*. Următorul program, *fmode.c*, afișează valoarea curentă a variabilei *_fmode*:

```
#include <stdio.h>
#include <fcntl.h>  //Contine declaratia variabilei _fmode
```

```
void main(void)
{
   if (_fmode == O_TEXT)
     printf("Interpretare de tip text \n");
   else
     printf("Interpretare de tip binar\n");
}
```

INTRAREA FILES= DIN CONFIG.SYS

C/C++ 367

Când lucrați într-un mediu DOS, intrarea *FILES* din fișierul *config.sys* precizează numărul de fișiere pe care sistemul le poate deschide simultan (Windows limitează numărul fișierelor deschise în funcție de memoria disponibilă a sistemului, de spațiul de pe disc, de utilizarea altor resurse și așa mai departe). Așa cum am mai spus în această secțiune, DOS utilizează primii cinci indicatori de fișiere pentru *stdin, stdout, stderr, stdaux* și *stdprn*. În mod prestabilit, DOS dispune de opt indicatori de fișiere. Cum acest număr este prea mic pentru programe (cu excepția celor foarte simple), majoritatea utilizatorilor măresc numărul de indicatori disponibili la 20 sau 30, ca mai jos:

```
FILES=30
```

Intrarea FILE definește numărul fișierelor care pot fi deschide de DOS – nu numărul fișierelor care pot fi deschise de fiecare program ce lucrează sub DOS. Când rulați programe rezidente în memorie, de exemplu, acestea pot să deschidă fișiere despre care să nu fiți informat. Dacă stabiliți intrarea *FILE* la un număr prea mare de indicatori (DOS acceptă până la 255 de indicatori), nu înseamnă că programele în C pot deschide atât de multe fișiere. Deschiderea unui număr mare de fișiere în cadrul programelor în C ridică două probleme. În primul rând, majoritatea compilatoarelor restricționează dimensiunea tabloului de pointeri de fișier la 20. Pentru a putea deschide mai mult de 20 de fișiere, trebuie să modificați dimensiunea tabloului. În al doilea rând, așa cum ați învățat, DOS restricționează la 20 numărul de fișiere pe care le poate deschide un program. Pentru a putea deschide mai mult de 20 de fișiere, trebuie să utilizați serviciile sistemului de operare DOS pentru a-i cere acestuia să accepte mai mult de 20 de fișiere deschise în programul dumneavoastră curent.

Observație: *Secțiunea 369 explică indicatorii de fișier.*

UTILIZAREA OPERAȚIILOR I/O CU FIȘIERE LA NIVEL JOS ȘI LA NIVEL ÎNALT

C/C++ 368

Atunci când programele dumneavoastră lucrează cu fișiere, ele pot executa două tipuri de operații de intrare sau de ieșire: de *nivel jos* și de *nivel înalt*. În toate secțiunile prezentate până în acest moment, au fost utilizate capacitățile de nivel înalt ale limbajului C (bazate pe flux), cum ar fi *fopen*, *fgets* sau *fputs*. Atunci când folosiți funcțiile de nivel înalt pentru operații I/O cu fișiere ale limbajului C, acestea, la rândul lor, utilizează serviciile sistemului de operare, care sunt bazate pe *indicatori de fișiere*. Biblioteca run-time a limbajului C dispune de funcții de nivel jos, pe care le puteți utiliza în programele dumneavoastră. În loc să lucreze cu un pointer de flux, funcțiile de nivel jos utilizează *descriptori de fișier*. Tabelul 368 descrie pe scurt câteva dintre cele mai utilizate funcții de nivel jos ale limbajului C.

Nume	Scop
close	Închide fișierul asociat indicatorului de fișier precizat, golind bufferele fișierului
creat	Creează un fișier pentru operații de ieșire, returnând un indicator de fișier
open	Deschide un fișier existent pentru operații I/O, returnând un indicator de fișier
read	Citește un număr precizat de octeți din fișierul asociat indicatorului de fișier dat
write	Scrie un număr precizat de octeți în fișierul asociat indicatorului de fișier dat

***Tabelul 368** Funcții de nivel jos ale limbajului C.*

Când vă scrieți programele, alegerea între utilizarea funcțiilor de nivel jos sau a funcțiilor de nivel înalt depinde de preferința dumneavoastră. Totuși, rețineți că cei mai mulți programatori înțeleg mai bine modul de utilizare a funcțiilor de nivel înalt ale limbajului C. Ca urmare, dacă folosiți funcții de nivel înalt, cum ar fi *fopen* sau *fgets*, codul dumneavoastră va fi înțeles de mai mulți programatori.

369 INDICATORII DE FIȘIER



După cum știți, intrarea *FILES* din fișierul *config.sys* vă permite să specificați numărul indicatorilor de fișier pe care îi acceptă DOS. Pe scurt, un *indicator de fișier* este o valoare întreagă care definește în mod unic un fișier deschis. Atunci când folosiți funcții de nivel jos ale limbajului C pentru operații I/O cu fișiere, veți declara indicatorul de fișier al programului dumneavoastră ca fiind de tip *int*, ca mai jos:

```
int indicator_intrare, indicator_iesire;
```

Funcțiile *open* și *creat* returnează descriptori de fișier sau valoarea –1 dacă nu poate fi deschis fișierul:

```
int fisier_nou, fisier_vechi;
fisier_nou = creat("FISIER.NOU", S_IWRITE);  // Creeaza un
                                              // fisier nou
                                              // pentru iesire
fisier_vechi = open("FISIER.OLD", O_RDONLY); // Deschide un
                                              // fisier existent
                                              // pentru citire
```

Sistemul DOS atribuie fiecărui fișier pe care îl deschideți sau îl creați un indicator unic de fișier. Valoarea indicatorului este, de fapt, un index în tabela cu fișierele de proces, cu ajutorul căreia DOS ține evidența fișierelor pe care le deschide programul.

370 TABELA CU FIȘIERELE DE PROCES



Atunci când rulați un program în mediul DOS, acesta ține evidența fișierelor pe care le deschide programul utilizând *tabela cu fișierele de proces*. În cadrul prefixului de segment al programului, sistemul DOS păstrează un pointer *far* la tabela care descrie fișierele deschise ale programului. De fapt, tabela conține intrări într-o a doua tabelă, *tabela fișierelor din sistem*, în cadrul căreia sistemul DOS urmărește toate fișierele deschise. Figura 370 ilustrează relațiile dintre indicatorul de fișier, tabela cu fișierele de proces și tabela fișierelor din sistem.

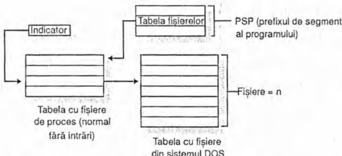


Figura 370 *Relațiile dintre indicatorul de fișier, tabela cu fișierele de proces și tabela cu fișierele din sistem*

Sub Windows, Task Manager întreține lista tuturor proceselor deschise, iar Windows utilizează tabela fișierelor din sistemul DOS pentru a întreține lista tuturor fișierelor deschise. Compact-discul care însoțește această carte include programul *Task_Man.cpp*, care prezintă lista tuturor programelor deschise la un moment dat în sistem.

VIZUALIZAREA INTRĂRILOR ÎN TABELA CU FIȘIERELE DE PROCES

C/C++ 371

Așa cum se spune în secțiunea 370, sistemul DOS utilizează tabela cu fișiere de proces pentru a urmări fișierele deschise de program. La deplasamentul 18H, în cadrul prefixului de segment al programului, se află un tablou de valori întregi. Valorile cuprinse în acest tablou specifică indexurile tabelei de fișiere din sistemul DOS. Dacă o valoare nu este folosită, sistemul de operare o stabilește la FFH (255 în sistemul zecimal). Următorul program, *tab_fis.c*, va afișa valorile din tabela cu fișierele de proces. Amintiți-vă că această tabelă conține valori întregi care servesc ca indexuri în cadrul tabelei cu fișierele din sistem:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>
void main(void)
  {
    struct fcbs
      {
        char drive;
        char filename[8];
        char extension[3];
        int current_block;
        int record_size;
      };
    typedef struct fcbs fcb;
    struct program_segment_prefix
      {
        char near *int20;
```

```
      char near *next_paragraph_segment;
      char reserved_1;
      char dos_dispatcher[5];
      char far *terminate_vector;
      char far *ctrlc_vector;
      char far *critical_error_vector;
      char near *parent_psp;
      unsigned char file_table[20];
      char near *environment_block_segment;
      char far *stack_storage;
      int handles_available;
      char far *file_table_address;
      char far *shares_previous_psp;
      char reserved_2[20];
      char dos_int21_retf[3];
      char reserved_3[9];
      fcb fcb1;
      fcb fcb2;
      char reserved_4[4];
      char command_tail[128];
    } far *psp;
  int i;
  psp = (struct program_segment_prefix far *) ((long)_psp << 16);
  for (i = 0; i < 20; i++)
    printf("Intrarea %d contine %x\n", i, psp->file_table[i]);
}
```

Atunci când compilați și executați programul *tab_fis.c*, veți vedea că sunt folosite primele cinci intrări în tabela cu fișierele de proces. Aceste intrări corespund funcțiilor *stdin, stdout, stderr, stdaux* și *stdprn*. Editați acest program și deschideți unul sau mai multe fișiere înainte de a afișa intrările în tabela cu fișiere și veți găsi mai multe intrări în cadrul tabelei cu fișiere de proces.

372 TABELA FIȘIERELOR DIN SISTEM



Indicatoarele de fișier sunt valori de index în cadrul tabelei cu fișierele de proces, care, la rândul său, trimite la tabela cu fișierele din sistem. Tabela cu fișierele din sistem păstrează informații despre orice fișier deschis de sistemul DOS, de un driver de dispozitiv, de un program rezident în memorie sau de programele dumneavoastră. Figura 372 prezintă conținutul tabelei cu fișierele de proces.

De fapt, DOS împarte tabela sistemului în două secțiuni. Prima secțiune conține cinci intrări. A doua dispune de spațiu atât cât este necesar pentru numărul de intrări pe care îl specificați cu intrarea FILES din fișierul *config.sys* (mai puțin cinci – intrările care se păstrează în prima secțiune).

Offset	Câmp
00H	Pointer far la următoarea tabelă
04H	Număr de intrări în această tabelă
06H	Indicatori la această intrare
08H	Mod de deschidere a fișierului
0AH	Atribut fișier
0BH	Dispozitiv local/la distanță
0DH	Antet driver sau DPB
12H	Cluster de start
14H	Marcare oră
16H	Marcare dată
18H	Dimensiune fișier
1CH	Deplasament pointer curent
20H	Cluster relativ
22H	Sector intrare director
26H	Deplasament intrare director
27H	Nume și extensie fișier
34H	Rezervat
44H	

Figura 372 *Conținutul tabelei cu fișierele din sistem*

AFIȘAREA TABELEI CU FIȘIERELE DIN SISTEM

C/C++ 373

În cadrul tabelei cu fișierele din sistem, DOS păstrează informații despre fiecare fișier deschis. Utilizând lista DOS cu tipurile de liste, prezentată în secțiunea despre DOS și BIOS a acestei cărți, următorul program, *tab_sis.c*, afișează intrările tabelei cu fișierele din sistem:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

void main(void)
  {
    union REGS inregs, outregs;
    struct SREGS segs;
    int i, j;
    int dim_struct;
    struct SystemTableEntry
      {
        struct SystemTableEntry far *next; //Urmatoarea intrare SFT
        unsigned file_count;     //Fisiere in tabela
        unsigned handle_count;   //Indicatori la acest fisier
        unsigned open_mode;      //Mod deschidere fisier
        char file_attribute;     //Octet atribut
        unsigned local_remote;   //Validare bit 15 inseamna
                                 // la distanta
        unsigned far *DPD;       //Bloc parametri unitate
        unsigned starting_cluster;

        unsigned time_stamp;
```

```
      unsigned date_stamp;
      long file_size;
      long current_offset;
      unsigned relative_cluster;
      long directory_sector_number;
      char directory_entry_offset;
      char filename_ext[11]; // Fara punct, completat cu spatii
                             // Se ignora campurile SHARE
    } far *ptr_tabel, far *fis;
    long far *tabela_sistem;
//Se obtine versiunea DOS
    inregs.x.ax = 0x3001;
    intdos (&inregs, &outregs);
    if (outregs.h.al < 3)
      {
        printf("Programul cere versiunea DOS 3 sau superioara\n");
        exit (1);
      }
    else if (outregs.h.al == 3)
      dim_struct = 0x35;
    else if (outregs.h.al >= 4)
      dim_struct = 0x3B;
      //Se obtine lista cu pointerii listei
    inregs.h.ah = 0x52;
    intdosx (&inregs, &outregs, &segs);
      // Pointerul la tabela cu fisiere din sistem e la
      // deplasamentul 4
    tabela_sistem = MK_FP(segs.es, outregs.x.bx + 4);
    ptr_tabel = (struct SystemTableEntry far *) *tabela_sistem;
    do {
      printf("%d intrari in tabela\n", ptr_tabel->file_count);
      for (i = 0; i < ptr_tabel->file_count; i++)
        {
          fis = MK_FP(FP_SEG(ptr_tabel), FP_OFF(ptr_tabel) +
            (i * dim_struct));
          if (fis->handle_count)
            {
              for (j = 0; j < 8; j++)
                if (fis->filename_ext[j] != ' ')
                  putchar(fis->filename_ext[j]);
                else
                  break;
              if (fis->filename_ext[8] != ' ')
                putchar('.');
              for (j = 8; j < 11; j++)
                if (fis->filename_ext[j] != ' ')
                  putchar(fis->filename_ext[j]);
```

```
            printf(" %ld octeti %d referinte %x atribute\n",
            fis->file_size, fis->file_attribute),
              fis->handle_count;
           }
         }
       ptr_tabel = ptr_tabel->next;
     } while (FP_OFF(ptr_tabel) != 0xFFFF);
 }
```

Când rulați programul *tab_sis.c* la prompterul *DOS*, ieșirea pe ecran nu este atât de interesantă. Dacă lucrați însă cu Windows, lansați-l și folosiți pictograma MS-DOS pentru deschiderea ferestrei DOS. Din interiorul ferestrei DOS, rulați programul *tab_sis.c*. Puteți, de asemenea, să editați programul și să utilizați funcția *fopen* pentru a deschide unul sau mai multe fișiere înainte de afișarea conținutului tabelei cu fișierele de sistem.

OBȚINEREA INDICATORILOR DE FIȘIER DIN POINTERII DE FLUX

C/C++ 374

Secțiunea 360 a prezentat structura *FILE* definită în fișierul antet *stdio.h*. Ați învățat că atunci când executați operații de nivel înalt cu fișiere utilizând funcțiile *fopen* sau *fgets*, declarați pointeri de flux cu structura *FILE*, ca mai jos:

```
FILE *intrare, *iesire;
```

Funcțiile limbajului C convertesc apoi pointerii de flux în indicatori de fișier pentru executarea operațiilor I/O propriu-zise. Pentru a înțelege mai bine relația dintre pointerii de flux și indicatorii de fișier, analizați următorul program, *indic.c*, care deschide fișierul *config.sys* din directorul rădăcină și apoi îi afișează descriptorul de fișier, precum și indicatorii de fișier *stdin*, *stdout*, *stderr*, *stdaux* și *stdprn*:

```
#include <stdio.h>

void main(void)
  {
   FILE *intrare;

   if ((intrare = fopen("\\CONFIG.SYS", "r")) == NULL)
     printf("Eroare la deschidere \\CONFIG.SYS\N");
   else
     {
      printf("Indicator pentru CONFIG.SYS %d\n", intrare->fd);
      printf("Indicator pentru stdin %d\n", stdin->fd);
      printf("Indicator pentru stdout %d\n", stdout->fd);
      printf("Indicator pentru stderr %d\n", stderr->fd);
      printf("Indicator pentru stdaux %d\n", stdaux->fd);
      printf("Indicator pentru stdprn %d\n", stdprn->fd);
      fclose(intrare);
     }
  }
```

Atunci când compilaţi şi executaţi programul *indic.c*, pe ecran vor fi afişate valorile indicatorilor între 0 şi 5.

375 SCRIEREA UNEI IEŞIRI FORMATATE ÎN FIŞIER

C/C++

Câteva secţiuni din acest capitol prezintă moduri în care programele dumneavoastră pot scrie într-un fişier. În multe cazuri, programele trebuie să formateze ieşirea spre fişier. De exemplu, dacă aţi creat un raport de inventar, veţi dori să îl aliniaţi pe coloane, să folosiţi text şi numere şi aşa mai departe. În primul capitol al acestei cărţi, „Primele noţiuni de C", aţi învăţat cum se foloseşte funcţia *printf* pentru a realiza ieşiri formatate pe ecran. În mod similar, limbajul C dispune de funcţia *fprintf*, care utilizează specificatori de format pentru a scrie ieşirea formatată într-un fişier, cum se vede mai jos:

```
#include <stdio.h>

int fprintf(FILE *pointer_fisier, const char *specif_format,
   [argument[,...]]);
```

Următorul program, *fprintf.c*, utilizează funcţia *fprintf* pentru a scrie o ieşire formatată într-un fişier numit *fprintf.dat*:

```
#include <stdio.h>

void main(void)
  {
    FILE *pointer_fisier;

    int pag = 800;
    float pret = 49.95;

    if (pointer_fisier = fopen("FPRINTF.DAT", "w"))
      {
        fprintf(pointer_fisier, "Totul despre C/C++\n");
        fprintf(pointer_fisier, "Pagini: %d\n", pag);
        fprintf(pointer_fisier, "Pretul: $%5.2f\n", pret);
        fclose(pointer_fisier);
      }
    else
     printf("Eroare la deschiderea FPRINTF.DAT\n");
  }
```

376 REDENUMIREA FIŞIERELOR

C/C++

Întrucât programele dumneavoastră lucrează cu fişiere, uneori poate va trebui să redenumiţi sau să mutaţi un fişier. Pentru asemenea cazuri, limbajul C dispune de funcţia *rename*. Formatul funcţiei *rename* este următorul:

```
#include <stdio.h>

int rename(const char *nume_vechi, const char *nume_nou);
```

Dacă reușește redenumirea sau mutarea fișierului, funcția *rename* returnează valoarea 0. Dacă apare o eroare, funcția va returna o valoare diferită de zero și va atribui variabilei globale *errno* una dintre valorile stării de eroare prezentate în tabelul 376.

Valoare	Semnificație
EACCES	Acces interzis
ENOENT	Fișierul nu a fost găsit
EXDEV	Nu se poate muta de pe un disc pe altul

Tabelul 376 *Valorile stării de eroare pentru funcția* ***rename.***

Următorul program, *renum.c*, utilizează funcția *rename* pentru a crea un program care poate redenumi sau muta fișierul specificat în linia de comandă:

```
#include <stdio.h>

void main(int argc, char *argv[])
  {
    if (argc < 3)
      printf("Trebuie precizat numele fisierului sursa si tinta\n");
    else if (rename(argv[1], argv[2]))
      printf("Eroare la redenumirea fisierului\n");
  }
```

Observație: *Secțiunea 1472 prezintă în detaliu felul în care puteți redenumi un fișier utilizând interfața Windows API.*

ȘTERGEREA UNUI FIȘIER



Când programele dumneavoastră lucrează cu fișiere, uneori va trebui să ștergeți unul sau mai multe fișiere. În asemenea cazuri, programele dumneavoastră în C pot utiliza funcția *remove*. Formatul funcției *remove* este următorul:

```
#include <stdio.h>

int remove(const char *nume_fisier);
```

Dacă ștergerea fișierului se realizează cu succes, funcția va returna valoarea 0. Dacă apare o eroare, funcția *remove* va returna valoarea –1 și va atribui variabilei globale *errno* una dintre valorile prezentate în tabelul 377.

Valoare	Semnificație
EACCES	Acces interzis
ENOENT	Fișierul nu a fost găsit

Tabelul 377 *Erorile pe care le întoarce funcția* ***remove***.

Următorul program, *sterg.c*, utilizează funcția *remove* pentru a șterge toate fișierele specificate în linia de comandă:

```
#include <stdio.h>

void main(int argc, char *argv[])
```

```
   {
     while (*++argv)
       if (remove(*argv))
         printf("Eroare la stergere %s\n", *argv);
   }
```

Pe lângă funcţia *remove*, majoritatea compilatoarelor de C oferă funcţia *unlink*, care şterge, de asemenea, fişiere:

```
#include <io.h>

int unlink(const char *nume_fisier);
```

Dacă funcţia *unlink* reuşeşte ştergerea fişierului, ea va returna valoarea 0. Dacă apare o eroare, funcţia va returna eroarea de stare –1 şi va atribui variabilei globale *errno* una dintre constantele de eroare prezentate în tabelul 377. Următorul program, *unlink.c*, utilizează funcţia *unlink* pentru a şterge fişierele specificate în linia de comandă:

```
#include <stdio.h>

void main(int argc, char *argv[])
   {
     while (*++argv)
       if (unlink(*argv))
         printf("Eroare la stergere %s\n", *argv);
   }
```

Observaţie: *Secţiunea 1473 prezintă în detaliu felul în care puteţi şterge un fişier utilizând interfaţa Windows API.*

378 DETERMINAREA MODULUI ÎN CARE UN PROGRAM POATE ACCESA UN FIŞIER



Atunci când programele dumneavoastră lucrează cu fişiere, va trebui uneori să determinaţi dacă programul poate accesa un fişier. Funcţia limbajului C denumită *access* testează dacă există un anumit fişier şi dacă puteţi deschide acest fişier în modul pe care îl doriţi. Formatul funcţiei *access* este următorul:

```
#include <io.h>

int access(const char *nume_fisier, int mod_acces);
```

Parametrul *mod_acces* specifică modul în care programul dumneavoastră are nevoie să deschidă fişierul, aşa cum este arătat în tabelul 378.1.

Valoare	Semnificație
0	Testează dacă fișierul există
2	Testează dacă se poate scrie în fișier
4	Testează dacă fișierul poate fi citit
6	Testează dacă programul are permisiunea de a scrie și citi fișierul

Tabelul 378.1 *Valorile pe care le poate lua parametrul **mod_acces.***

Dacă programul poate accesa fișierul în modul specificat, funcția *access* va returna valoarea 0. Dacă apare o eroare, funcția va returna –1 și va atribui variabilei globale *errno* una dintre valorile de eroare prezentate în tabelul 378.2.

Valoare	Semnificație
EACCES	Acces interzis
ENOENT	Fișierul nu a fost găsit

Tabelul 378.2 *Erorile pe care le returna funcția **access**.*

Următorul program, *access.c*, utilizează funcția *access* pentru a determina modul în care programul dumneavoastră poate accesa fișierul specificat în linia de comandă:

```
#include <stdio.h>
#include <io.h>
void main(int argc, char *argv[])
  {
   int mod_acces;
   mod_acces = access(argv[1], 0);
   if (mod_acces)
    printf("Fisierul %s nu exista\n");
   else
     {
      mod_acces = access(argv[1], 2);
      if (mod_acces)
       printf("Fisierul nu poate fi scris\n");
      else
       printf("Fisierul poate fi scris\n");
      mod_acces = access(argv[1], 4);
      if (mod_acces)
       printf("Fisierul nu poate fi citit\n");
      else
       printf("Fisierul poate fi citit\n");
      mod_acces = access(argv[1], 6);
      if (mod_acces)
       printf("Fisierul nu poate fi citit/scris\n");
      else
       printf("Fisierul poate fi citit/scris\n");
     }
  }
```

Observație: *Secțiunea 1462 prezintă în detaliu utilizarea atributelor de fișier în mediul Windows pentru determinarea modului în care un program poate accesa un fișier.*

379 STABILIREA MODULUI DE ACCES AL UNUI FIȘIER



Atunci când programele dumneavoastră lucrează cu fișiere, uneori veți dori să modificați drepturile de acces pentru scriere și citire ale unui program. De exemplu, să presupunem că aveți un fișier care conține date importante. Pentru a proteja fișierul atunci când programul nu rulează, puteți configura fișierul numai cu drepturi de citire. În acest mod, utilizatorul nu poate șterge accidental fișierul. Atunci când începe programul, puteți modifica drepturile de acces pentru scriere și citire după necesități. În asemenea cazuri, programele dumneavoastră pot utiliza funcția *chmod*, ca mai jos:

```
#include <sys\stat.h>
#include <io.h>

int chmod(const char *nume_fisier, int mod_acces);
```

Fișierul antet *sys\stat.h* definește constantele modului de acces, așa cum sunt prezentate în tabelul 379.1.

Valoare	Semnificație
S_IWRITE	Permisiunea de citire este autorizată
S_IREAD	Permisiunea de scriere este autorizată

Tabelul 379.1 *Constantele modului de acces pentru* ***chmod.***

Pentru a dispune de acces la scriere și citire, efectuați o operație *SAU pe biți* asupra celor două constante (*S_IWRITE* | *S_IREAD*). Dacă funcția *chmod* a modificat cu succes atributele fișierului, va returna valoarea 0. Dacă apare o eroare, funcția va returna –1 și va atribui variabilei globale *errno* una dintre valorile de eroare prezentate în tabelul 379.2.

Valoare	Semnificație
ENOENT	Fișierul nu a fost găsit
EACCES	Acces interzis

Tabelul 379.2 *Erorile pe care le returnează funcția* ***chmod.***

Următorul program, *readonly.c*, stabilește fișierul specificat în linia de comandă cu accesul exclusiv pentru citire:

```
#include <stdio.h>
#include <sys\stat.h>
#include <io.h>

void main(int argc, char *argv[])
  {
    if (chmod(argv[1], S_IREAD))
      printf("Eroare la setare %s\n", argv[1]);
  }
```

Observație: *Secțiunea 1462 prezintă în detaliu utilizarea atributelor de fișier în mediul Windows pentru schimbarea modului în care un program poate accesa un fișier.*

OBȚINEREA UNUI CONTROL MAI BUN ASUPRA ATRIBUTELOR DE FIȘIER

C/C++ 380

În secțiunea 379, ați învățat cum se utilizează funcția *chmod* a limbajului C pentru a stabili atributele de scriere și citire ale unui fișier. Atunci când utilizați sistemul de operare DOS, puteți lucra cu atributele prezentate în tabelul 380.1.

Valoare	Semnificație
FA_ARCH	Atribut de arhivă
FA_DIREC	Atribut de director
FA_HIDDEN	Atribut de fișier ascuns
FA_LABEL	Etichetă de volum de disc
FA_RDONLY	Atribut de citire exclusivă
FA_SYSTEM	Atribut de sistem

Tabelul 380.1 *Atributele de fișier pe care le puteți utiliza în cadrul sistemului de operare DOS.*

Observație: *Unele compilatoare folosesc alte denumiri pentru aceste constante. Dacă vreți să aflați numele corect al constantelor, examinați fișierul* ***dos.h****, livrat împreună cu compilatorul dumneavoastră.*

Pentru a vă ajuta să lucrați cu aceste atribute, unele compilatoare pun la dispoziție funcția *_chmod*, al cărei format este arătat mai jos:

```
#include <dos.h>
#include <io.h>

int _chmod(const char *nume_fisier, int operatie [, int atribut]);
```

Parametrul *operatie* informează funcția dacă doriți să stabiliți sau să obțineți valoarea atributelor. Dacă funcția apelantă stabilește acest parametru la 0, funcția *_chmod* returnează atributele fișierului curent. Dacă funcția apelantă stabilește parametrul la 1, funcția *_chmod* validează atributul specificat. Parantezele drepte arată că parametrul *atribut* este opțional. Dacă funcția *_chmod* se execută cu succes, ea va returna atributele fișierului curent. Dacă apare o eroare, funcția *_chmod* returnează valoarea –1 și va atribui variabilei globale *errno* una dintre valorile prezentate în tabelul 380.2.

Valoare	Semnificație
ENOENT	Fișierul nu a fost găsit
EACCES	Acces interzis

Tabelul 380.2 *Erorile pe care le returnează funcția* ***_chmod***.

Următorul program, *atrib.c*, utilizează funcția *_chmod* pentru afișarea atributelor fișierului curent:

```
#include <stdio.h>
#include <dos.h>
```

```
#include <io.h>
void main(int argc, char *argv[])
  {
    int atribut;
    if ((atribut = _chmod(argv[1], 0)) == -1)
      printf("Eroare la accesare %s\n", argv[1]);
    else
      {
        if (atribut & FA_ARCH)
          printf("Arhiva ");
        if (atribut & FA_DIREC)
          printf("Director ");
        if (atribut & FA_HIDDEN)
          printf("Ascuns ");
        if (atribut & FA_LABEL)
          printf("Eticheta volum ");
        if (atribut & FA_RDONLY)
          printf("Numai pentru citit ");
        if (atribut & FA_SYSTEM)
          printf("Sistem ");
      }
  }
```

Multe compilatoare dispun, de asemenea, de funcțiile *_dos_getfileattr* și *_dos_setfileattr*, care permit obținerea sau stabilirea atributelor DOS ale fișierelor, ca mai jos:

```
#include <dos.h>
int _dos_getfileattr(const char *nume_fisier, unsigned atribut);
int _dos_setfileattr(const char *nume_fisier, unsigned atribut);
```

Funcțiile *_dos_getfileattr* și *_dos_setfileattr* utilizează constantele de atribut prezentate în tabelul 380.3.

Valoare	Semnificație
_A_ARCH	Atribut de arhivă
_A_HIDDEN	Atribut de fișier ascuns
_A_NORMAL	Atribut normal
_A_RDONLY	Atribut de citire exclusivă
_A_SUBDIR	Atribut de director
_A_ SYSTEM	Atribut de sistem
_A_VOLID	Etichetă de volum de disc

Tabelul 380.3 *Constantele de atribut pe care utilizează funcțiile* ***_dos_getfileattr*** *și* ***_dos_setfileattr***.

Dacă *_dos_getfileattr* și *_dos_setfileattr* se execută cu succes, ele returnează valoarea 0. Dacă apare o eroare, funcțiile returnează valoare –1 și atribuie variabilei globale *errno* valoarea *ENOENT* (fișierul nu a fost găsit).

De regulă, programele trebuie să lucreze numai cu atributele de arhivă, de citire exclusivă și de fișier ascuns, rezervând celelalte atribute pentru a fi folosite de sistemul DOS. Dacă schimbați numai atributul de citire exclusivă, utilizați funcția *chmod*, prezentată în secțiunea 379, pentru a mări portabilitatea programului dumneavoastră.

***Observație:** Secțiunea 1463 prezintă în detaliu utilizarea atributelor de fișier sub Windows pentru schimbarea modului în care programul poate accesa fișierele.*

TESTATREA ERORILOR DE FLUX

C/C++ 381

Atunci când programele dumneavoastră execută operații I/O cu un fișier, trebuie să testeze întotdeauna valoarea returnată de funcțiile *fopen*, *fputs*, *fgets* și celelalte, pentru a verifica dacă operația a reușit. Pentru a vă ajuta să executați aceste testări, limbajul C dispune de macroinstrucțiunea *ferror*, care examinează un flux I/O pentru a detecta erorile de citire sau scriere. Dacă apare o eroare, macroinstrucțiunea *ferror* returnează valoarea adevărat (*true*). Dacă nu a apare, macroinstrucțiunea returnează fals (*false*), cum se vede mai jos:

```
#include <stdio.h>
int ferror(FILE *flux);
```

După apariția unei erori, macroinstrucțiunea *ferror* va rămâne adevărată până când programele dumneavoastră invocă macroinstrucțiunea *clearerr* pentru fluxul dat:

```
#include <stdio.h>
void clearerr(FILE *flux);
```

Următorul program, *ferror.c*, citește și afișează conținutul fișierului pe ecran. După fiecare operație I/O, programul testează existența erorilor. Dacă apare o eroare, programul se întrerupe și afișează un mesaj de eroare la *stderr*:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
  {
    FILE *pointer_fisier;
    char linie[256];
    if (pointer_fisier = fopen(argv[1], "r"))
      {
        while (fgets(linie, sizeof(linie), pointer_fisier))
          {
            if (ferror(pointer_fisier))
              {
                fprintf(stderr, "Eroare la citire %s\n", argv[1]);
                exit(1);
              }
```

```
          else
            {
              fputs(linie, stdout);
              if (ferror(pointer_fisier))
                {
                  fprintf(stderr, "Eroare la scriere in stdout\n");
                  exit(1);
                }
            }
        }
      }
    else
      printf("Eroare la deschidere %s\n", argv[1]);
  }
```

382 DETERMINAREA DIMENSIUNII UNUI FIȘIER



Atunci când programele dumneavoastră execută operații I/O cu fișiere, uneori este nevoie să determinați dimensiunea în octeți a unui fișier. Pentru astfel de ocazii, limbajul C vă oferă funcția *filelength*. Această funcție returnează o valoare de tip *long* și așteaptă un indicator de fișier, nu un pointer de fișier, așa cum se vede mai jos:

```
#include <io.h>
long filelength(int indicator_fisier);
```

Dacă se execută cu succes, funcția returnează *dimensiunea* în octeți a fișierului. Dacă apare o eroare, funcția *filelength* returnează valoarea –1 și stabilește variabila globală *errno* la *EBADF* (număr greșit de fișier). Următorul program, *filedim.c*, va afișa pe ecran dimensiunea unui fișier dat:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

void main(int argc, char *argv[])
  {
    int indicator_fisier;
    long dim_fisier;

    if ((indicator_fisier = open(argv[1], O_RDONLY)) == -1)
     printf("Eroare la deschiderea fisierului %d\n", argv[1]);
    else
      {
        dim_fisier = filelength(indicator_fisier);
        printf("Dimensiunea fisierului in octeti e %ld\n",
          dim_fisier);
        close(indicator_fisier);
```

```
    }
}
```

Observație: *Secțiunea1463 prezintă amănunțit felul în care veți determina dimensiunea unui fișier utilizând interfața Windows API.*

GOLIREA UNUI FLUX I/O

C/C++383

Pentru a crește nivelul de performanță al programelor dumneavoastră, biblioteca run-time a limbajului C reține în mod normal ieșirea fișierului până când se umple bufferul de scriere pe disc (în care, de obicei, încape un sector de disc) sau până când închideți fișierul. În acest fel, biblioteca run-time reduce numărul lentelor operații I/O cu discul. Din păcate, atunci când programele dumneavoastră utilizează un astfel de buffer, există riscul pierderii de date. Atunci când se execută o funcție cum ar fi *fputs* pentru a scrie o ieșire și funcția nu returnează o eroare, programul presupune că sistemul de operare a înregistrat corect datele pe disc. În realitate, datele pot rămâne tot în memoria calculatorului. Dacă utilizatorul închide calculatorul, el va pierde datele respective. Dacă aveți un program pentru care trebuie să vă asigurați că toate datele sunt scrise pe disc, puteți utiliza funcția *fflush* pentru a indica bibliotecii run-time să scrie pe disc datele bufferului din memorie. Formatul funcției *fflush* este următorul:

```
#include <stdio.h>
int fflush(FILE * flux_fisier);
```

Dacă funcția *fflush* se execută cu succes, ea va returna valoarea 0. Dacă apare o eroare, funcția *fflush* va returna constanta *EOF.* Următoarele instrucțiuni ilustrează modul de utilizare a funcției *fflush* pentru a goli bufferul fișierului pe disc după fiecare operație de ieșire:

```
while (fgets(linie, sizeof(linie), fisier_intrare))
    {
      fputs(linie, fisier_iesire);
      fflush(fisier_iesire);
    }
```

Observație: *Atunci când folosiți funcția **fflush**, cereți bibliotecii run-time a compilatorului de C să invoce un serviciu al sistemului de operare pentru a scrie datele pe disc. Dacă sistemul de operare utilizează un buffer propriu (numit **cache de disc**), sistemul de operare poate plasa datele în acest buffer, nu pe disc. În funcție de programul utilizat pentru memoria cache de disc, poate fi posibilă invocarea unui alt serviciu al sistemului de operare pentru a goli ieșirea.*

ÎNCHIDEREA TUTUROR FIȘIERELOR DESCHISE ÎNTR-UN SINGUR PAS

C/C++384

Așa cum s-a arătat în secțiunea 361, înainte ca programele dumneavoastră să se încheie, ar trebui să utilizați funcția *fclose* pentru a închide fișierele deschise. Să presupunem că aveți o funcție care execută o operație critică. Dacă funcția întâmpină o eroare, programul ar trebui închis imediat. Din păcate, funcția nu are capacitatea de a cunoaște ce fișiere sunt deschise.

În asemenea cazuri, programul dumneavoastră poate utiliza funcția *fcloseall*, care închide toate fișierele deschise, cum se vede mai jos:

```
#include <stdio.h>

int fcloseall(void);
```

Dacă funcția *fcloseall* se execută cu succes, ea va returna numărul de fișiere pe care a reușit să le închidă. Dacă apare o eroare, funcția *fcloseall* returna constanta *EOF*. Următoarele instrucțiuni ilustrează modul în care puteți utiliza funcția *fcloseall*:

```
if (eroare == CRITICAL)
   {
     fprintf(stderr, "Eroare critica de dispozitiv\n");
     fcloseall();
     exit(1);
   }
```

385 OBȚINEREA INDICATORULUI DE FIȘIER AL UNUI FLUX

C/C++

Așa cum s-a arătat în secțiunea 360, atunci când programele dumneavoastră execută operații cu fișiere, ele pot executa operații de nivel înalt utilizând fluxuri (*FILE *flux*). Puteți, de asemenea, să utilizați indicatorii de fișier de nivel jos (*int indicator*). Așa cum ați învățat, unele dintre funcțiile bibliotecii run-time a limbajului C cer indicatori de fișier. Dacă programul dumneavoastră utilizează fluxurile, puteți să închideți fișierul și să-l deschideți din nou utilizând indicatorul de fișier sau puteți să obțineți un indicator de fișier utilizând funcția *fileno*, cum se vede mai jos:

```
#include <stdio.h>

int fileno(FILE *flux);
```

Următorul program, *fileno.c*, utilizează funcția *fileno* pentru a obține un indicator de fișier pentru un flux deschis:

```
#include <stdio.h>
#include <io.h>

void main(int argc, char *argv[])
  {
    FILE *flux;
    int indicator;
    long dim_fisier;

    if (flux = fopen(argv[1], "r"))
      {
        // Instructiuni
        indicator = fileno(flux);
        dim_fisier = filelength(indicator);
        printf("Dimensiune fisier %ld\n", dim_fisier);
```

```
        fclose(flux);
      }
    else
      printf("Eroare la deschidere %s\n", argv[1]);
  }
```

CREAREA UNUI NUME DE FIȘIER TEMPORAR UTILIZÂND P_TMPDIR

C/C++ 386

Atunci când programele dumneavoastră execută operații de intrare/ieșire, trebuie adesea să deschidă unul sau mai multe fișiere temporare sau să scrie ieșirea într-un fișier care nu există pe disc. În asemenea cazuri, dificultatea provine din faptul că numele fișierului trebuie să fie unic, pentru ca programul să nu suprascrie un fișier existent. Pentru ca programele dumneavoastră să genereze nume unice, puteți utiliza funcția *tmpnam*, ca mai jos:

```
#include <stdio.h>

char *tmpnam(char *buffer);
```

Dacă programul transmite un buffer către *tmpnam*, funcția va atribui numele temporar bufferului. Dacă invocați funcția *tmpnam* cu *NULL*, ea va aloca memorie pentru numele fișierului și va returna un pointer la începutul numelui. Funcția *tmpnam* examinează intrarea *P_tmpdir* din fișierul antet *stdio.h*. Dacă *P_tmpdir* este definită, funcția *tmpnam* creează numele unic de fișier în directorul corespunzător. Altfel, funcția *tmpnam* va crea fișierul în directorul curent. Rețineți că funcția *tmpnam* nu creează de fapt fișierul, ci returnează un nume de fișier pe care programul dumneavoastră îl poate folosi cu funcțiile *fopen* sau *open*. Următorul program, *tmpnam.c*, ilustrează modul de utilizare a funcției *tmpnam*:

```
#include <stdio.h>

void main(void)
  {
    char buffer[64];
    int contor;

    for (contor = 0; contor < 5; contor++)
      printf("Numele temporar %s\n", tmpnam(buffer));
  }
```

Observație: *Compact-discul care însoțește această carte conține programul* ***creattmp.cpp****, care creează un fișier temporar cu interfața Windows API.*

CREAREA UNUI NUME DE FIȘIER TEMPORAR UTILIZÂND TMP SAU TEMP

C/C++ 387

Atunci când programele dumneavoastră execută operații de intrare/ieșire, trebuie adesea să deschidă unul sau mai multe fișiere temporare sau să scrie ieșirea într-un fișier care nu există pe disc. În asemenea cazuri, dificultatea provine din faptul că numele fișierului trebuie să fie

unic, pentru ca programul să nu suprascrie un fişier existent. Pentru ca programele dumneavoastră să genereze un nume unic, puteţi utiliza funcţia *tempnam*, ca mai jos:

```
#include <stdio.h>

char *tempnam(char *buffer, char *prefix);
```

Dacă programul transmite un buffer către *tempnam*, funcţia va atribui numele temporar bufferului. Dacă invocaţi funcţia *tempnam* cu *NULL*, ea va aloca memorie pentru numele fişierului şi va returna un pointer la începutul numelui de fişier. Parametrul *prefix* vă permite să definiţi un set de caractere pe care doriţi ca funcţia *tempnam* să le plaseze la începutul fiecărui nume de fişier. Funcţia *tempnam* examinează intrările de mediu pentru a determina dacă există o intrare TMP sau TEMP. Dacă este definită o intrare TMP sau TEMP, funcţia *tempnam* va crea numele unic de fişier în directorul corespunzător. Altfel, funcţia *tempnam* va crea fişierul în directorul curent. Reţineţi că funcţia *tempnam* nu creează de fapt fişierul, ci returnează un nume de fişier pe care programul dumneavoastră îl poate folosi cu funcţiile *fopen* sau *open*. Următorul program, *tempnam.c*, ilustrează modul de utilizare a funcţiei *tempnam*:

```
#include <stdio.h>

void main(void)
  {
    char buffer[64];
    int contor;
    printf("Numele temporar %s\n", tempnam(buffer, "Compendiu"));
  }
```

388 CREAREA UNUI FIŞIER TEMPORAR ADEVĂRAT



În secţiunile 386 şi 387, aţi învăţat cum se utilizează funcţiile *tmpnam* şi *tempnam* pentru a genera nume de fişier temporare. Aşa cum aţi învăţat, funcţiile *tmpnam* şi *tempnam* nu creează în realitate un fişier, ci, pur şi simplu, ele returnează un nume de fişier care nu se utilizează în mod curent. În plus, limbajul C oferă funcţia *tmpfile*, care determină un nume unic de fişier, deschide fişierul şi returnează un pointer de fişier programului. Veţi apela funcţia *tmpfile* aşa cum se vede mai jos:

```
#include <stdio.h>

FILE *tmpfile(void);
```

Dacă funcţia *tmpfile* se execută cu succes, ea deschide un fişier în modul citire/scriere şi returnează un pointer de fişier. Dacă apare o eroare, funcţia *tmpfile* întoarce *NULL*. Fişierul returnat de funcţia *tmpfile* este un fişier temporar. Când programul se încheie (sau apelează funcţia *rmtmp*), sistemul de operare şterge fişierul respectiv. Următoarea instrucţiune ilustrează modul în care programul dumneavoastră utilizează funcţia *tmpfile*:

```
FILE *fisier_temp;

  if (fisier_temp = tmpfile())
```

```
   {
      // Fisierul temporar se deschide cu succes
      // Instructiuni care utilizeaza fisierul
   }
else
   printf("Eroare la deschidere fisier temporar\n");
```

ELIMINAREA FIȘIERELOR TEMPORARE

C/C++ 389

În secțiunea 388, ați învățat că funcția *tmpfile* permite programelor dumneavoastră să creeze fișiere temporare, cu un conținut care nu există decât pe durata executării programului. În programul dumneavoastră, puteți să eliberați fișierele temporare înainte de încheierea programului. În astfel de cazuri, puteți folosi funcția *rmtmp*, al cărei format este următorul:

```
#include <stdio.h>
int rmtmp(void);
```

Dacă funcția se execută cu succes, ea returnează numărul de fișiere temporare pe care le-a închis și le-a șters.

CĂUTAREA UNUI FIȘIER ÎN COMANDA PATH

C/C++ 390

Când lucrați în mediul DOS și executați o comandă externă, comanda PATH definește directoarele în care sistemul DOS caută fișierele EXE, COM și BAT. Deoarece subdirectoarele definite în PATH conțin, în mod normal, comenzile cele mai frecvent utilizate, e posibil ca programul să caute un fișier în intrările subdirectoarelor din PATH. Pentru aceste cazuri, unele compilatoare furnizează funcția *searchpath*. Funcția este invocată specificând fișierul dorit ca argument. Dacă funcția *searchpath* reușește să localizeze fișierul, ea returnează numele de cale complet al fișierului pe care programul dumneavoastră îl poate deschide cu funcția *fopen*. Dacă funcția *searchpath* nu găsește fișierul, ea returnează *NULL*, ca mai jos:

```
#include <dir.h>
char *searchpath(const char *nume_fisier);
```

Următorul program, *searchpath.c*, ilustrează modul de utilizare a funcției *searchpath* pentru a căuta fișierul specificat:

```
#include <stdio.h>
#include <dir.h>
void main(int argc, char *argv[])
 {
   char *cale;
   if (cale = searchpath(argv[1]))
     printf("Nume cale: %s\n", cale);
   else
     printf("Fisierul nu a fost gasit\n");
 }
```

***Observație:** Funcția **searchpath** caută fișierul specificat în directorul curent și apoi, în subdirectoarele liniei de comandă PATH.*

391 CĂUTAREA UNUI FIȘIER ÎN LISTA CU SUBDIRECTOARELE INTRĂRII DE MEDIU

C/C++

În secțiunea 390, ați utilizat funcția *searchpath* pentru a căuta un anumit fișier în directoarele din comanda PATH. În mod similar, puteți să căutați un fișier în directoarele specificate în altă intrare de mediu. De exemplu, multe compilatoare de C definesc intrările *LIB* și *INCLUDE*, care precizează locația unor fișiere bibliotecă (cu extensia *.lib*) și fișiere antet. Pentru a căuta directoarele pe care le specifică intrările *LIB* și *INCLUDE*, puteți utiliza funcția *_searchenv*, cum se vede mai jos:

```
#include <dos.h>

char *_searchenv(const char *nume_fisier, const char
  *intrare_mediu, *nume_cale);
```

Funcția *_searchenv* caută numele de fișier precizat în directoarele specificate în *intrare_mediu*. Dacă funcția *_searchenv* găsește fișierul, ea va atribui numele de cale al fișierului unui buffer de tip șir de caractere și va returna un pointer la numele de cale. Dacă funcția *_searchenv* nu găsește fișierul, ea va returna *NULL*. Următorul program, *cautintr.c*, utilizează funcția *searchenv* pentru a căuta un anumit fișier în subdirectoarele specificate în intrarea *LIB*:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    char cale[128];

    _searchenv(argv[1], "LIB", cale);
    if (cale[0])
      printf("Nume cale: %s\n", cale);
    else
      printf("Fisierul nu a fost gasit\n");
  }
```

***Observație:** Funcția **_searchenv** caută fișierul specificat în directorul curent înainte de a căuta în subdirectoarele intrării de mediu.*

***Observație:** În secțiunile de la 1474 până la 1476, veți învăța cum se folosește interfața Windows API pentru găsirea fișierelor în sistem.*

392 DESCHIDEREA FIȘIERELOR ÎN DIRECTORUL TEMP

C/C++

După cum știți, multe programe își creează fișiere temporare în subdirectoarele specificate de intrarea de mediu TEMP în fișierul *config.sys*. În programele dumneavoastră, puteți să creați cu ușurință propriile fișiere temporare în directorul specificat de intrarea TEMP,

folosind funcţia *getenv*. Următoarele instrucţiuni ilustrează modul în care programele dumneavoastră pot deschide un fişier numit *tempdata.dat* în cadrul directorului temporar:

```
char nume_cale[_MAX_PATH];
strcpy(nume_cale, getenv("TEMP"));
if (nume_cale[0])
   strcat(nume_cale, "\\TEMPDATA.DAT");
else
   strcat(nume_cale, "TEMPDATA.DAT");
if (pointer_fisier = fopen(nume_cale, "w"))
```

În acest fragment de cod, dacă intrarea TEMP există, programul deschide fişierul în subdirectorul corespunzător. Dacă nu există intrarea TEMP, programul deschide fişierul temporar în directorul curent. Observaţi că fragmentul de cod presupune că variabila TEMP nu conţine o valoare care se încheie cu caracterul backslash. În mod ideal, programele dumneavoastră vor testa valoarea curentă a variabilei TEMP şi vor proceda în consecinţă.

MINIMIZAREA OPERAŢIILOR I/O CU FIŞIERE

C/C++ 393

Comparativ cu viteza de lucru a unităţii centrale şi a memoriei calculatorului dumneavoastră, discul mecanic este foarte lent. Drept urmare, trebuie să încercaţi să minimizaţi numărul operaţiilor I/O cu discul, executate de programele dumneavoastră. În ceea ce priveşte operaţiile cu fişiere, deschiderea unui fişier consumă, probabil, cel mai mult timp. De aceea, trebuie ca întotdeauna să examinaţi programele dumneavoastră pentru a fi siguri că nu vor deschide şi închide fişiere în mod inutil sau că nu deschide în mod repetat un fişier în cadrul unei bucle. De exemplu, să considerăm următoarele instrucţiuni:

```
while (optiune_meniu != QUIT)
  {
    if (pointer_fisier = fopen("DATABASE.DAT", "r"))
      {
        // Obtinerea numelui clientului
        client(nume);
        // Cauta informatii client in fisier
        caut_client_info(nume, pointer_fisier, date_buffer);
        fclose(pointer_fisier);
      }
    else
      {
        eroare_deschis_fisier("Renunta...");
      }
    optiune_meniu = alege_optiune_meniu();
  }
```

Instrucţiunile se execută în mod repetat, furnizând informaţii despre client până când utilizatorul selecţionează opţiunea QUIT. Reţineţi că apelarea funcţiei *fopen* are loc în cadrul buclei. De aceea, programul execută în mod repetat operaţii I/O cu discul. Pentru a îmbunătăţi performanţele sistemului, funcţia *fopen* trebuie scoasă în afara buclei. Dacă

funcția *caut_client_info* trebuie să pornească de la începutul fișierului, programul poate să repoziționeze pointerul, cum se vede mai jos:

```
if (pointer_fisier = fopen("DATABASE.DAT", "r"))
   eroare_deschis_fisier("Renunta...");
while (optiune_meniu != QUIT)
   {
     // Obtinerea numelui clientului
     client(nume);
     rewind(pointer_fisier);  // Repozitioneaza pointerul la
                              // inceput fisier
     // Cauta informatii client in fisier
     caut_client_info(nume, pointer_fisier, date_buffer);
     optiune_meniu = alege_optiune_meniu();
   }
fclose(pointer_fisier);
```

394 SCRIEREA UNUI COD CARE UTILIZEAZĂ CARACTERUL BACKSLASH ÎN NUMELE DIRECTOAARELOR



În câteva dintre secțiunile acestui capitol, s-a lucrat cu nume de directoare. De exemplu, funcția *chdir* permite programelor dumneavoastră să selecteze un anumit director. Când programele dumneavoastră precizează numele directorului ca o valoare constantă, asigurați-vă că ați folosit caracterul backslash dublu (\\) în cadrul numelui de cale, potrivit cerințelor. Următoarea apelare a funcției *chdir*, de exemplu, încearcă să selecteze subdirectorul DOS:

```
status = chdir("\DOS");
```

Când folosiți caracterul backslash în cadrul unui șir de caractere în C, amintiți-vă că limbajul C tratează acest caracter ca pe un simbol special. Atunci când compilatorul de C întâlnește caracterul backslash, el testează caracterul care urmează pentru a determina dacă este un simbol special și, în caz afirmativ, îl înlocuiește cu echivalentul său din ASCII. Dacă după *backslash* urmează un caracter care nu este un simbol special, compilatorul ignoră caracterul backslash. De aceea, funcția anterioară *chdir* încearcă selectarea directorului DOS, nu \\DOS. Invocarea corectă a funcției *chdir* în acest caz ar trebui să fie următoarea:

```
status = chdir("\\DOS");
```

395 SCHIMBAREA DIRECTORULUI CURENT



La executarea programelor dumneavoastră, uneori poate fi necesară schimbarea directorului curent. Pentru această operație, cele mai multe compilatoare de C dispun de funcția *chdir*. Funcția *chdir* este foarte asemănătoare cu comanda DOS CHDIR. Dacă invocați funcția *chdir* cu un șir de caractere care nu conține litera unității de disc, ea va căuta directorul pe unitatea

curentă. Următoarea apelare a funcţiei *chdir*, de exemplu, selectează directorul *data* de pe unitatea de disc C:

```
status = chdir("C:\\DATA"); // Observati ca se foloseste \\
```

În mod similar, următoarea comandă selectează directorul *tclite* de pe unitatea de disc curentă:

```
status = chdir("\\TCLITE");
```

Dacă funcţia *chdir* se execută cu succes, ea va returna valoarea 0. Dacă directorul nu există, funcţia *chdir* va returna valoarea –1 şi va stabili variabila globală *errno* la constanta *ENOENT*. Următorul program, *chdir.c*, implementează comanda DOS CHDIR:

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <errno.h>

void main(int argc, char *argv[])
  {
    char director[MAXPATH];

    if (argc == 1) // Afiseaza directorul curent
      {
        getcwd(director, MAXPATH);
        puts(director);
      }
    else if ((chdir(argv[1])) && (errno == ENOENT))
      puts("Director nevalabil");
  }
```

Observaţie: *În locul simbolului **MAXPATH**, unele compilatoare definesc simbolul **_MAX_PATH** în fişierul antet **direct.h** (sau **dir.h**).*

Observaţie: *Secţiunea 1468 explică în detaliu schimbarea directoarelor sub Windows.*

CREAREA UNUI DIRECTOR

C/C++396

La executarea programelor dumneavoastră, uneori poate fi necesară crearea unui director. Pentru aceasta, cele mai multe compilatoare de C dispun de funcţia *mkdir*. Funcţia *mkdir* este foarte asemănătoare cu comanda DOS MKDIR. Dacă invocaţi funcţia *mkdir* cu un şir de caractere care nu conţine litera unităţii de disc, ea va crea directorul pe unitatea curentă. Următoarea apelare a funcţiei, de exemplu, creează directorul DATA pe unitatea C:

```
status = mkdir("C:\\DATA"); // Observati ca se foloseste \\
```

În acelaşi mod, următoarea comandă creează directorul TEMPDATA pe unitatea curentă, în directorul curent:

```
status = mkdir("TEMPDATA");
```

Dacă funcția *mkdir* se execută cu succes, ea va returna valoarea 0. Dacă însă ea nu va putea crea directorul, va returna valoarea –1.

Observație: *Secțiunea 1467 explică în detaliu crearea directoarelor sub Windows.*

397 ȘTERGEREA UNUI DIRECTOR



La executarea programelor dumneavoastră, uneori poate fi necesară crearea sau eliminarea unui director. Pentru a facilita ștergerea unui director în programele dumneavoastră, cele mai multe compilatoare de C dispun de funcția *rmdir*. Funcția *rmdir* este foarte asemănătoare cu comanda DOS RMDIR. Dacă invocați funcția *mkdir* cu un șir de caractere care nu conține litera unității de disc, ea va crea directorul pe unitatea curentă. Următoarea apelare a funcției, de exemplu, șterge directorul DATA de pe unitatea C:

```
status = rmdir("C:\\DATA"); // Observati ca se foloseste \\
```

Într-un mod asemănător, următoarea comandă șterge directorul TEMPDATA de pe unitatea și directorul curente:

```
status = rmdir("TEMPDATA");
```

Dacă funcția *rmdir* se execută cu succes, ea va returna valoarea 0. Dacă însă directorul nu există sau funcția *rmdir* nu poate să îl șteargă, ea va returna valoarea –1 și va atribui variabilei globale *errno* una dintre valorile prezentate în tabelul 397.

Valoare	Semnificație
EACCES	Acces interzis
ENOENT	Nu există un asemenea director

Tabelul 397 *Valorile de eroare pentru funcția* ***rmdir.***

Observație: *Secțiunea 1470 explică în detaliu ștergerea directoarelor sub Windows.*

398 ȘTERGEREA UNUI ARBORE DE DIRECTOARE



În versiunea 6 a sistemului de operare MS-DOS, Microsoft a introdus comanda DELTREE. Comanda DELTREE permite ștergerea într-un singur pas a unui director împreună cu fișierele și subdirectoarele din interiorul său. Dacă nu folosiți versiunea 6 a sistemului MS-DOS, puteți să creați propria dumneavoastră comandă DELTREE utilizând programul *deltree.c*, ca mai jos:

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <alloc.h>
#include <string.h>

void main(int argc, char **argv)
  {
    void sterge_arbore(void);
```

```
    char buffer[128];
    char unitate[MAXDRIVE], director[MAXDIR], numefisier[MAXFILE],
      ext[MAXEXT];
    if (argc < 2)
      {
       printf("Eroare de sintaxa\n");
       exit(0);
      }
    fnsplit (argv[1], unitate, director, numefisier, ext);
    getcwd (buffer, sizeof(buffer));
    if (unitate[0] == NULL)
      {
       fnsplit (buffer, unitate, director, numefisier, ext);
       strcpy (buffer, director);
       strcat (buffer, numefisier);
       strcat (buffer, ext);
      }
    else
      {
       printf("Litera unitatii de disc nespecificata\n");
       exit (1);
      }
    if (strcmpi(buffer, argv[1]) == 0)
      {
       printf("Nu se poate sterge directorul curent\n");
       exit (1);
      }
    getcwd (director, 64);
    if (chdir (argv[1]))
     printf("Director nevalabil %s\n", argv[1]);
    else
     sterge_arbore();
    chdir (director);
    rmdir (argv[1]);
  }
union REGS inregs, outregs;
struct SREGS segs;
void sterge_arbore(void)
  {
    struct ffblk fileinfo;
    int rezultat;
    char far *farbuff;
    unsigned dta_seg, dta_ofs;
    rezultat = findfirst("*.*", &fileinfo, 16);
    inregs.h.ah = 0x2f;
```

```
    intdosx (&inregs, &outregs, &segs);
    dta_seg = segs.es;
    dta_ofs = outregs.x.bx;
    while (! rezultat)
      {
        if ((fileinfo.ff_attrib & 16) &&
            (fileinfo.ff_name[0] != '.'))
          {
            inregs.h.ah = 0x1A;
            inregs.x.dx = FP_SEG(farbuff);
            segread(&segs);
            intdosx (&inregs, &outregs, &segs);
            chdir (fileinfo.ff_name);
            sterge_arbore();
            chdir ("..");
            inregs.h.ah = 0x1A;
            inregs.x.dx = dta_ofs;
            segs.ds = dta_seg;
            rmdir (fileinfo.ff_name);
          }
        else if (fileinfo.ff_name[0] != '.')
          {
            remove (fileinfo.ff_name);
          }
        rezultat = findnext (&fileinfo);
      }
  }
```

Observație: *Compact-discul care însoțește această carte conține fișierul* ***win_dtree.cpp****, care execută aceeași funcție ca și programul* ***deltree.c****, dar lucrează sub Windows 95 sau Windows NT.*

399 CONSTRUIREA UNUI NUME DE CALE ÎNTREG



Atunci când programele dumneavoastră lucrează cu fișiere și cu directoare, pot apărea situații în care trebuie să cunoașteți numele de cale întreg al unui fișier. De exemplu, dacă directorul curent este *data*, iar unitatea de disc curentă este C, numele de cale întreg al fișierului *raport.dat* este *c:\data\raport.dat*. Pentru găsirea numelui de cale întreg al unui fișier (ceea ce înseamnă combinarea componentelor căii), unele compilatoare de C dispun de o funcție numită *fnmerge*. Funcția folosește cinci parametri: un buffer în cadrul căruia funcția plasează numele de cale întreg, numele unității de disc, numele directorului, numele fișierului și extensia, cum se vede mai jos:

```
#include <dir.h>

void fnmerge(char *buffer, const char *unitate,
             const char *director, const char *numefisier,
             const char *extensie);
```

Dacă parametrul *buffer* ia valoarea *NULL*, funcția *fnmerge* va aloca memoria folosită pentru a păstra numele de cale întreg. Dacă funcția *fnmerge* va reuși să obțină numele de cale întreg, va returna un pointer la buffer. Dacă apare o eroare, funcția returnează *NULL*. Următorul program, *numecale.c*, ilustrează modul de utilizare a funcției *fnmerge*:

```
#include <string.h>
#include <stdio.h>
#include <dir.h>

void main(void)
  {
   char s[MAXPATH];
   char unitate[MAXDRIVE];
   char director[MAXDIR];
   char fisier[MAXFILE];
   char ext[MAXEXT];

   getcwd(s, MAXPATH); /* Obtine directorul de lucru curent */
   strcat(s, "\\"); /* Adauga la cale caracterul \ */
   fnsplit(s, unitate, director, fisier, ext);
          /* Imparte sirul in elemente separate */
   strcpy(fisier, "DATA");
   strcpy(ext, ".TXT");
   fnmerge(s, unitate, director, fisier, ext);
         /* combina toate elementele intr-un sir */
   puts(s); /* afiseaza sirul rezultat */
  }
```

Observație: *Unele compilatoare utilizează fișierul antet* ***direct.h*** *în loc de* ***dir.h***.

ANALIZAREA CĂII UNUI DIRECTOR

C/C++ 400

Atunci când programele dumneavoastră lucrează cu fișiere și directoare, pot apărea situații în care trebuie să împărțiți numele de cale, separând litera unității de disc, calea subdirectorului, numele fișierului și extensia. Pentru a vă ajuta să analizați numele de cale (adică să o separați în componentele sale), unele compilatoare dispun de funcția *_splitpath*. Formatul apelării funcției este următorul:

```
#include <dir.h>
int _splitpath(const char *cale, const char *unitate,
               const char *director, const char *numefisier,
               const char *extensie);
```

Următorul program, *split.c*, ilustrează modul de utilizare a funcției *_splitpath*:

```
#include <stdio.h>
#include <direct.h>
#include <stdlib.h>

void main(void)
  {
    char *path_1 = "C:\\SUBDIR\\NUMEFIS.EXE";
    char *path_2 = "SUBDIR\\NUMEFIS.EXE";
    char *path_3 = "NUMEFIS.EXE";
    char subdir[MAXDIR];
    char unitate[MAXDRIVE];
    char numefisier[MAXFILE];
    char extensie[MAXEXT];
    int flags;        // Pastreaza valoarea returnata de fnsplit
    flags = fnsplit (cale_1, unitate, subdir, numefisier, extensie);
    printf("Imparte %s\n", cale_1);
    printf("Unitate %s Subdir %s Fisier %s Extensie %s\n",
      unitate, subdir, numefisier, extensie);
    flags = fnsplit (cale_2, unitate, subdir, numefisier, extensie);
    printf("Imparte %s\n", cale_2);
    printf("Unitate %s Subdir %s Fisier %s Extensie %s\n",
      unitate, subdir, numefisier, extensie);
    flags = fnsplit (cale_3, unitate, subdir, numefisier, extensie);
    printf("Imparte %s\n", cale_3);
    printf("Unitate %s Subdir %s Fisier %s Extensie %s\n",
      unitate, subdir, numefisier, extensie);
  }
```

Observați utilizarea constantelor pentru definirea corectă a dimensiunilor bufferului. Atunci când compilați și executați programul *split.c*, pe ecran vor fi afișate următoarele:

```
Imparte C:\SUBDIR\NUMEFIS.EXE
Unitate C: Subdir \SUBDIR\ Fisier NUMEFIS Extensie .EXE
Imparte \SUBDIR\NUMEFIS.EXE
Unitate Subdir \SUBDIR\ Fisier NUMEFIS Extensie .EXE
Imparte NUMEFIS.EXE
Unitate Subdir Fisier NUMEFIS Extensie .EXE
C:\>
```

401 CONSTRUIREA UNUI NUME DE CALE



Atunci când programele dumneavoastră lucrează cu fișiere și directoare, pot apărea situații în care trebuie să combinați litera unității de disc, subdirectorul, numele fișierului și extensia într-un nume de cale complet. Pentru a efectua o astfel de operație, unele compilatoare dispun de funcția *fnmerge*. Formatul funcției este următorul:

```
fnmerge(numecale, unitate, subdir, numefisier, extensie);
```

Următorul program, *numcale.c*, ilustrează modul de utilizare a funcției *fnmerge*:

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>

void main(void)
  {
    char numecale[MAXPATH];
    char *unitate = "C:";
    char *subdir = "\\SUBDIR";
    char *numefisier = "NUMEFIS";
    char *extensie = "EXT";

    fnmerge(numecale, unitate, subdir, numefisier, extensie);
    printf("Numele de cale complet este %s\n", numecale);
  }
```

Atunci când compilați și executați programul *numcale.c*, pe ecran va fi afișat următorul rezultat:

```
Numele de cale complet este C:\SUBDIR\NUMEFIS.EXT
C:\>
```

DESCHIDEREA ȘI ÎNCHIDEREA UNUI FIȘIER UTILIZÂND FUNCȚII DE NIVEL JOS

C/C++ 402

Limbajul C permite operații I/O cu fișiere de nivel înalt (care lucrează cu fluxuri) și de nivel jos (care lucrează cu intervale de octeți). Atunci când executați operații I/O cu fișiere de nivel jos, puteți deschide un fișier existent utilizând funcția *open*. Pentru a închide apoi fișierul, puteți utiliza funcția *close*, așa cum se vede mai jos:

```
#include <fcnt.h>
#include <sys\stat.h>

int open(const char *cale, int mod_acces [,mod_creare]);
int close(int indicator);
```

Dacă funcția *open* reușește să deschidă fișierul, ea va returna indicatorul acestuia. Dacă apare o eroare, funcția *open* va returna valoarea –1 și va stabili variabila globală *errno* la una dintre valorile prezentate în tabelul 402.1.

Valoare	Semnificație
ENOENT	Fișierul nu a fost găsit
EMFILE	Prea multe fișiere deschise
EACCES	Acces interzis
EINVACC	Cod de acces nevalabil

Tabelul 402.1 *Codurile stărilor de eroare pe care funcția* ***open*** *le atribuie variabilei globale* ***errno.***

Parametrul *cale* este un şir de caractere care conţine numele fişierului dorit. Parametrul *mod_acces* specifică modul în care vreţi să folosiţi fişierul. Valoarea *mod_creare* poate fi o combinaţie (utilizează operaţia *SAU pe biţi*) a valorilor prezentate în tabelul 402.2.

Mod de acces	Semnificaţie
O_RDONLY	Acces numai pentru citire
O_WRONLY	Acces numai pentru scriere
O_RDWR	Acces pentru scriere şi citire
O_NDELAY	Utilizează valoare de întârziere UNIX
O_APPEND	Poziţionează pointerul pentru operaţii de adăugare
O_TRUNC	Trunchiază conţinutul unui fişier existent
O_EXCL	Dacă *O_CREAT* este specificat şi fişierul există deja, funcţia *open* returnează o eroare
O_BINARY	Deschide un fişier în modul binar
O_TEXT	Deschide un fişier în modul text

Tabelul 402.2 *Valorile posibile pentru parametrul* ***mod_acces*** *atunci când utilizaţi funcţia* ***open.***

În mod prestabilit, funcţia *open* nu va crea o ieşire dacă fişierul nu există. Dacă doriţi ca funcţia *open* să creeze fişiere, trebuie să includeţi indicatorul *O_CREAT* înaintea modului de acces dorit (de exemplu, *O_CREAT | O_TEXT*). Dacă specificaţi *O_CREAT*, puteţi folosi parametrul *mod_creare* pentru a preciza modul în care doriţi să creaţi fişierul. Parametrul *mod_creare* poate folosi o combinaţie a valorilor prezentate în tabelul 402.3.

Mod de creare	Semnificaţie
S_IWRITE	Creat pentru operaţii de scriere
S_IREAD	Creat pentru operaţii de citire

Tabelul 402.3 *Valorile acceptate de funcţia* ***open*** *pentru parametrul* ***mod_creare.***

Următoarea instrucţiune ilustrează modul de folosire a funcţiei *open* pentru deschiderea fişierului *config.sys* din directorul rădăcină, exclusiv pentru operaţii de citire:

```
if ((indicator = open("\\CONFIG.SYS", O_RDONLY)) == -1)
  printf("Eroare la deschidere fisierului \\ CONFIG.SYS\n");
else
  // Instructiuni
```

Dacă doriţi să deschideţi fişierul *iesire.dat* pentru operaţii de scriere şi vreţi ca funcţia *open* să creeze un fişier care nu există, folosiţi funcţia *open*, ca mai jos:

```
if ((indicator = open("\\CONFIG.SYS", O_RDONLY | O_CREAT,
    S_IWRITE)) == -1)
  printf("Eroare la deschidere fisierului \\ CONFIG.SYS\n");
else
  // Instructiuni
```

Când funcția termină de utilizat fișierul, îl puteți închide folosind funcția *close*, cum se vede mai jos:

```
close(indicator)
```

CREAREA UNUI FIȘIER C/C++ 403

În secțiunea 402, ați învățat că, în mod prestabilit, funcția *open* nu creează un fișier care nu există. Ați învățat, de asemenea, că puteți indica funcției *open* să creeze un fișier atunci când specificați *O_CREAT* la modul de acces. Dacă utilizați un compilator mai vechi, funcția *open* nu acceptă *O_CREAT*. Prin urmare, veți avea nevoie să utilizați funcția *creat*, cum se vede mai jos:

```
#include <sys\stat.h>

int creat(const char *cale, int mod_creare);
```

Parametrul *cale* specifică fișierul pe care vreți să îl creați. Parametrul *mod_creare* poate să conțină o combinație a valorilor prezentate în tabelul 403.

Mod de creare	Semnificație
S_IWRITE	Creat pentru operații de scriere
S_IREAD	Creat pentru operații de citire

***Tabelul 403** Valorile posibile pentru parametrul **mod_creare**.*

Dacă funcția *creat* se execută cu succes, ea va returna indicatorul fișierului. Dacă apare o eroare, funcția *creat* va returna valoarea –1 și va atribui o valoare de stare de eroare variabilei globale *errno*. Modul de conversie (binar sau text) pe care îl folosește funcția *creat* depinde de valoarea variabilei globale *_fmode*. Dacă fișierul cu numele specificat există deja, funcția *creat* va trunchia conținutul fișierului. Următoarea instrucțiune ilustrează modul în care se utilizează funcția *creat* pentru crearea fișierului *iesire.dat*:

```
if ((indicator = creat("IESIRE.DAT", S_IWRITE)) == -1)
  printf("Eroare la crearea fisierului\n");
else
  // Instructiuni
```

***Observație:** Dacă doriți să fie evident pentru un alt programator că ați creat un fișier, puteți să utilizați funcția **creat** în locul funcției **open** cu adăugarea **O_CREAT**.*

EXECUTAREA OPERAȚIILOR DE CITIRE ȘI DE SCRIERE DE NIVEL JOS C/C++ 404

Atunci când utilizați indicatori de fișier pentru a executa operații I/O cu fișiere de nivel jos, deschideți și închideți fișierele utilizând funcțiile *open* și *close*. În mod similar, puteți citi și scrie fișiere utilizând funcțiile *read* și *write*, cum arătăm în continuare:

```
#include <io.h>

int read(int indicator, void *buffer, unsigned lung);
int write(int indicator, void *buffer, unsigned lung);
```

Parametrul *indicator* este indicatorul returnat de funcțiile *open* sau *creat*. Parametrul *buffer* este fie bufferul din care funcția *read* citește informațiile, fie cel în care funcția *write* scrie date. Parametrul *lung* precizează numărul de octeți transferat de funcțiile *read* sau *write* (maximum 65534). Dacă funcția *read* reușește, ea va returna numărul de octeți citiți. Dacă întâlnește sfârșitul fișierului, ea va returna 0. În cazul unei erori, funcția *read* va returna –1 și va stabili variabila globală *errno* la una dintre valorile prezentate în tabelul 404.

Valoare	Semnificație
EACCES	Acces nevalabil
EBADF	Indicator de fișier nevalabil

Tabelul 404 *Valorile posibile de eroare pe care le returnează funcția* ***read.***

Dacă funcția *write* reușește, ea va returna numărul de octeți scriși. Dacă apare o eroare, ea va returna valoarea –1 și va atribui variabilei globale *errno* una dintre valorile arătate anterior. Următoarea buclă ilustrează modul în care puteți utiliza funcțiile *read* și *write* pentru a copia conținutul unui fișier în altul:

```
while (octeti_cititi = read(intrare, buffer, sizeof(buffer)))
   write(iesire, buffer, octeti_cititi);
```

405 TESTAREA SFÂRȘITULUI DE FIȘIER



În secțiunea 404, ați învățat că funcția *read* returnează valoarea 0 atunci când întâlnește *EOF*. În programul dumneavoastră, puteți să testați dacă s-a întâlnit sfârșitul de fișier înainte de a efectua o anumită operație. Atunci când utilizați indicatori de fișier, funcția *eof* returna valoarea 1 dacă pointerul de fișier a atins sfârșitul fișierului, 0 dacă pointerul nu este la sfârșitul fișierului și –1 dacă indicatorul de fișier este nevalabil:

```
#include <io.h>

int eof(int indicator);
```

Următoarele instrucțiuni modifică codul prezentat în secțiunea 404, pentru a utiliza *eof* la testarea sfârșitului fișierului de intrare:

```
while (! eof(intrare))
   {
     octeti_cititi = read(intrare, buffer, sizeof(buffer));
     write(iesire, buffer, octeti_cititi);
   }
```

406 UTILIZAREA RUTINELOR DE NIVEL JOS PENTRU OPERAȚII I/O CU FIȘIERE



Câteva secțiuni din acest capitol au abordat rutinele I/O de nivel jos destinate fișierelor. Pentru a vă ajuta să înțelegeți mai bine utilitatea fiecărei rutine, studiați următorul program, *jcopi.c*, care utilizează funcțiile *read* și *write* pentru a copia conținutul primului fișier specificat în linia de comandă în cel de al doilea:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

void main(int argc, char *argv[])
  {
    int sursa, destinatie; // Indicatori de fisiere
    char buffer[1024]; // Bufferul I/O
    int octeti_cititi;

    if (argc < 3)
      fprintf(stderr, "Trebuie specificate fisierele sursa si
                     destinatie\n");
    else if ((sursa = open(argv[1], O_BINARY | O_RDONLY)) == -1)
      fprintf(stderr, "Eroare la deschiderea %s\n", argv[1]);
    else if ((destinatie = open(argv[2], O_WRONLY | O_BINARY |
      O_TRUNC
        | O_CREAT, S_IWRITE)) == -1)
      fprintf(stderr, "Eroare la deschiderea %s\n", argv[2]);
    else
      {
        while (!eof(sursa))
          {
            if ((octeti_cititi= read(sursa, buffer,
              sizeof(buffer))) <= 0)
              fprintf(stderr, "Eroare la citirea fisierului sursa");
            else if (write(destinatie, buffer, octeti_cititi) !=
                octeti_cititi)
              fprintf(stderr, "Eroare la scrierea in fisierul
                      destinatie");
          }
        close(sursa);
        close(destinatie);
      }
  }
```

SPECIFICAREA MODULUI DE CONVERSIE A UNUI INDICATOR DE FIȘIER

C/C++ 407

Așa cum ați învățat, limbajul C traduce conținutul unui fișier, utilizând fie conversia de tip binar, fie pe cea de tip text. Dacă nu specificați altceva, limbajul C va utiliza valorile din variabila globală *_fmode* (*O_BINARY* sau *O_TEXT*) pentru a determina tipul conversiei. Atunci când deschideți sau creați un fișier utilizând rutinele de nivel jos ale limbajului C, puteți specifica modul de conversie a fișierului. În unele cazuri, programul dumneavoastră trebuie să specifice modul de conversie după ce deschideți fișierul. Pentru a specifica modul, puteți utiliza funcția *setmode*, ca mai jos:

```
#include <fcntl.h>

int setmode(int indicator, int mod_conversie);
```

Dacă execuția reușește, funcția returnează precedentul mod de conversie. Dacă apare o eroare, funcția *setmode* returnează valoarea –1 și stabilește variabila globală *errno* la *EINVAL* (argument nevalabil). Următoarea instrucțiune, de exemplu, stabilește fișierul asociat indicatorului *iesire* la conversia în modul de tip text:

```
if ((mod_vechi = setmode(iesire, O_TEXT)) == -1)
  printf("Eroare la stabilirea modului fisierului\n");
```

408 POZIȚIONAREA POINTERULUI DE FIȘIER UTILIZÂND LSEEK



Atunci când lucrați cu funcțiile limbajului C de manipulare a fișierelor la nivel jos, puteți să poziționați pointerul de fișier la o anumită locație în cadrul fișierului înaintea efectuării unei operații de citire sau scriere. Pentru a realiza aceasta, puteți utiliza funcția *lseek*, ca mai jos:

```
#include <io.h>

long lseek(int indicator, long deplasament, int relativ_la);
```

Parametrul *indicator* specifică pointerul de fișier pe care doriți să-l poziționați. Parametrii *deplasament* și *relativ_la* se combină pentru a specifica poziția dorită. Parametrul *deplasament* conține deplasamentul octetului din fișier. Parametrul *relativ_la* specifică locația din fișier începând de la care funcția *lseek* trebuie să aplice deplasamentul. Tabelul 408 prezintă valorile pe care le puteți utiliza pentru parametrul *relativ_la*.

Constantă	Semnificație
SEEK_CUR	De la poziția curentă din fișier
SEEK_SET	De la începutul fișierului
SEEK_END	De la sfârșitul fișierului

Tabelul 408 *Pozițiile de la care funcția* ***lseek*** *poate aplica un deplasament*

De exemplu, pentru a poziționa pointerul de fișier la sfârșitul unui fișier, veți utiliza funcția *lseek* ca mai jos:

```
lseek(indicator, 0, SEEK_END); // La sfarsitul fisierului
```

Dacă funcția reușește, ea va returna valoarea 0. Dacă apare o eroare, funcția *lseek* va returna o valoare diferită de 0.

409 DESCHIDEREA A MAI MULT DE 20 DE FIȘIERE



Așa cum ați învățat, un indicator de fișier este o valoare întreagă care identifică un fișier deschis. În realitate, un indicator de fișier este un index în tabela fișierelor de proces, care conține intrări pentru numai 20 de fișiere. Dacă programul dumneavoastră DOS trebuie să deschidă mai mult de 20 de fișiere, cea mai simplă soluție este să utilizați serviciile DOS de

fișier. Pentru a începe, programul dumneavoastră trebuie să ceară acceptarea a mai mult de 20 de fișiere. Pentru a crește numărul de indicatori de fișier, puteți utiliza funcția DOS *67H* din *INT 21H*. DOS va aloca, în acest caz, o tabelă suficient de mare pentru a cuprinde numărul respectiv de indicatori (până la 255 minus numărul de indicatori aflați curent în funcțiune). Apoi, programul dumneavoastră trebuie să deschidă fișierele utilizând serviciile DOS, nu biblioteca *run-time* a limbajului C. În acest fel, se poate evita limita de fișiere impusă de compilator. Următorul fragment de cod crește numărul de indicatori de fișier la 75:

```
inregs.h.ah = 0x67;
inregs.x.bx = 75; // Numar de indicatori
indos(&inregs, &outregs);

if (outregs.x.ax)
   printf("Eroare la alocarea indicatorilor\n");
```

Observație: *Numărul de indicatori de fișier disponibili este o problemă numai în mediul DOS sau într-o fereastră DOS. Windows determină limita numărului de fișiere pe care le puteți deschide la un moment dat în funcție de memoria curentă, spațiul liber pe hard-disc și alte considerente specifice.*

FOLOSIREA SERVICIILOR DOS DE FIȘIER

C/C++ 410

Așa cum detaliază secțiunea despre DOS și BIOS, DOS pune la dispoziție o colecție de servicii care vă permit deschiderea, citirea, scrierea și închiderea fișierelor. Pentru facilitarea utilizării acestor servicii de către limbajul C, multe compilatoare de C dispun de funcțiile prezentate în tabelul 410.

Funcție	Scop
_dos_creat	Creează un fișier și returnează indicatorul fișierului respectiv
_dos_close	Închide fișierul specificat
_dos_open	Deschide un fișier și returnează indicatorul fișierului respectiv
_dos_read	Citește numărul specificat de octeți din fișier
_dos_write	Scrie numărul specificat de octeți din fișier

Tabelul 410 *Funcții care utilizează serviciile de sistem DOS pentru fișiere.*

Pentru a înțelege mai bine serviciile de fișier, să considerăm următorul program, *doscopy.c*, care copiază conținutul primului fișier specificat în linia de comandă în cel de al doilea fișier:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
  {
    char buffer[1024];
    int intrare, iesire; // Indicatori de fisier
    unsigned octeti_cititi, octeti_scrisi;
             // Numar octeti transferati
    if (argc < 3)
```

```
    fprintf(stderr, "Trebuie specificate fisierul sursa si
                     destinatie\n");
  else if (_dos_open (argv[1], O_RDONLY, &intrare))
    fprintf(stderr, "Eroare la deschiderea fisierului sursa\n");
  else if (_dos_creat (argv[2], 0, &iesire))
    fprintf(stderr, "Eroare la deschiderea fisierului
     destinatie\n");
  else
   {
     while (!_dos_read(intrare, buffer, sizeof(buffer),
                      & octeti_cititi))
      {
        if (octeti_cititi == 0)
          break;
        _dos_write(iesire, buffer, octeti_cititi, &
           octeti_scrisi);
      }
     _dos_close(intrare);
     _dos_close(iesire);
   }
}
```

***Observație:** Deși rutinele DOS pentru fișiere sunt aproape similare cu funcțiile de nivel jos pentru fișiere din C, veți crește portabilitatea programelor dacă veți utiliza funcțiile C **open**, **read** și **write** în locul funcțiilor DOS. Cele mai multe compilatoare de C conțin suport pentru funcțiile de nivel jos.*

***Observație:** Când programați în Windows, pentru a gestiona fișierele, veți folosi funcțiile Windows API, nu rutinele DOS pentru fișiere. Secțiunile 1450-1478 prezintă în detaliu numeroase funcții Windows API pentru fișiere.*

411 OBȚINEREA MĂRCII OREI ȘI DATEI FIȘIERULUI C/C++

Atunci când executați listarea unui director, comanda DOS DIR va afișa numele fiecărui fișier, extensia, dimensiunea și data și ora la care fișierul a fost creat sau modificat ultima oară. Stocarea în fișier a datei și a orei de sistem DOS poartă denumirea de *marca datei și orei* fișierului. DOS modifică marca datei și a orei numai atunci când faceți modificări în fișier. Unele sisteme de operare, pe de altă parte, înregistrează data și ora la care fișierul a fost creat sau modificat ultima oară, precum și data și ora când fișierul a fost utilizat (citit) ultima oară. Sistemele de operare numesc *timpul ultimului acces* această a doua marcă a datei și a timpului. În funcție de scopul programului dumneavoastră, pot apărea situații în care trebuie să cunoașteți marca datei și a orei. În consecință, cele mai multe compilatoare pun la dispoziție funcția *_dos_getftime*, ca mai jos:

```
#include <dos.h>
unsigned _dos_getftime(ind indicator, unsigned *campdata,
                       unsigned *campora);
```

Dacă funcţia reuşeşte, ea returnează valoarea 0. Dacă apare o eroare, funcţia returnează o valoare diferită de 0 şi atribuie variabilei globale *errno* valoarea *EBADF* (indicator nevalabil). Parametrul *indicator* este un indicator de fişier deschis către fişierul dorit. Parametrii *campdata* şi *campora* reprezintă pointeri la valori întregi fără semn, la nivel de bit, cum apar în tabelele 411.1 şi, respectiv, 411.2.

Biţi pentru dată	Semnificaţie
0 – 4	Ziua de la 1 la 31
5 – 8	Luna de la 1 la 12
9 – 15	Anul de la 1980

Tabelul 411.1 *Componentele parametrului* ***campdata.***

Biţi pentru oră	Semnificaţie
0 – 4	Secunde împărţite la 2 (de la 1 la 30)
5 – 10	Minute de la 1 la 60
11 – 15	Ore de la 1 la 12

Tabelul 411.2 *Componentele parametrului* ***campora.***

Următorul program, *fisierdo.c*, utilizează funcţia *_dos_getftime* pentru a afişa marca datei şi a orei fişierului specificat în linia de comandă:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
  {
    unsigned data, ora;
    int indicator;

    if (_dos_open(argv[1], O_RDONLY, &indicator))
      fprintf(stderr, "Eroare la deschiderea fisierului sursa\n");
    else
     {
      if (_dos_getftime(indicator, &data, &ora))
        printf("Eroare la redarea marcii datei/orei\n");
      else
        printf("%s ultima modificare %02d-%02d-%d
          %02d:%02d:%02d\n", argv[1],
          (data & 0x1E0) >> 5, /* luna */
          (data & 0x1F), /* ziua */
          (data >> 9) + 1980, /* anul */
          (ora >> 11), /* ora */
          (ora & 0x7E0) >> 5, /* minutul */
          (ora & 0x1F) * 2); /* secunda */
        _dos_close(indicator);
     }
  }
```

După cum puteți vedea, programul utilizează operatorii pe biți ai limbajului C pentru a extrage câmpurile *ora* și *data*. În secțiunea 380, ați învățat cum se efectuează procese similare utilizând structuri cu câmpuri de biți.

Observație: *Secțiunea 1465 prezintă în detaliu felul în care veți obține marca datei și orei unui fișier în Windows.s*

412 OBȚINEREA DATEI ȘI OREI UNUI FIȘIER, UTILIZÂND CÂMPURI DE BIȚI



În secțiunea 411 ați utilizat funcția *_dos_getftime* pentru a obține marca datei și a orei unui fișier. Așa cum ați învățat, funcția *_dos_getftime* codifică câmpurile *data* și *ora* ca biți în cadrul a două valori *unsigned*. Pentru a extrage valorile câmpului, programul *fisierdo.c* utilizează operatorii pe biți ai limbajului C. Pentru a face programul dumneavoastră mai ușor de înțeles, puteți utiliza câmpurile de *biți* în cadrul unei structuri. Pentru aceasta, puteți utiliza următorul program, *dobiti.c*:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
{
    struct Data
    {
     unsigned int ziua:5;
     unsigned int luna:4;
     unsigned int anii:7;
    } data;
    struct Time
    {
     unsigned secunde:5;
     unsigned minute:6;
     unsigned ore:5;
    } time;
    int indicator;

    if (_dos_open(argv[1], O_RDONLY, &indicator))
    fprintf(stderr, "Eroare la deschiderea fisierului sursa\n");
    else
     {
     if (_dos_getftime(indicator, &data, &ora))
        printf("Eroare la obtinerea marcii datei si orei\n");
     else
        printf("%s ultima modificare %02d-%02d-
           %d %02d:%02d:%02d\n",
              argv[1],
              data.luna,          // luna
              data.ziua,          // ziua
```

```
                data.anii+ 1980,     // anul
                time.ore,            // ora
                time.minute,         // minute
                time.secunde * 2);   // secunde
            _dos_close(indicator);
        }
  }
```

Utilizând câmpurile de biți nu mai este necesar ca ceilalți programatori să înțeleagă operațiile complicate pe biți care au apărut în programul *fisierdo.c.*

Observație: *Secțiunea 1465 detaliază modul în care veți obține marca datei și orei unui fișier în mediu Windows.*

STABILIREA MĂRCII DATEI ȘI OREI UNUI FIȘIER — C/C++ 413

În secțiunile 411 și 412 ați utilizat funcția *_dos_getftime* pentru a obține marca datei și a orei unui fișier. În programul dumneavoastră, puteți să stabiliți marca datei și orei unui fișier. Pentru asemenea cazuri, majoritatea compilatoarelor de C dispun de funcția *_dos_setftime*, arătată mai jos:

```
#include <dos.h>

unsigned _dos_setftime(ind indicator, unsigned data,
  unsigned time);
```

Dacă funcția reușește, ea va returna valoarea 0. Dacă apare o eroare, funcția va returna o valoare diferită de 0. Parametrul *indicator* este un indicator către un fișier deschis. Parametrii *data* și *time* conțin valorile datei și orei codificate pe biți (la fel ca în secțiunea 411). Următorul program, *iulie4_97.c*, stabilește marca datei și a orei unui fișier, la amiaza zilei de 4 iulie 1997:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
  {
    union
      {
        struct Data
          {
            unsigned int ziua:5;
            unsigned int luna:4;
            unsigned int anii:7;
          } biti;
        unsigned val;
      } data;
    union
      {
```

```
        struct Time
          {
            unsigned secunde:5;
            unsigned minute:6;
            unsigned ore:5;
          } biti;
        unsigned val;
      } time;
    int indicator;

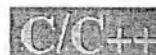
    if (_dos_open(argv[1], O_RDONLY, &indicator))
      fprintf(stderr, "Eroare la deschiderea fisierului sursa\n");
    else
      {
        data.biti.ziua = 4;
        data.biti.luna = 7;
        data.biti.anii = 17;      // 1980 + 17
        time.biti.ore = 12;
        time.biti.minute = 0;
        time.biti.secunde = 0;

        if (_dos_setftime(indicator, data.val,time.val))
          printf("Eroare la stabilirea datei/orei\n");
        _dos_close(indicator);
      }
}
```

Programul *iulie4_97.c* utilizează câmpurile de biți pentru a simplifica atribuirea biților datei și orei. Însă, funcția *_dos_setftime* cere parametri de tip *unsigned int*. Deoarece biții trebuie să fie văzuți în două moduri diferite, sunt foarte potriviți pentru o uniune (*union*). Secțiunea 481 prezintă în detaliu uniunile.

Observație: *Secțiunea 1465 detaliază modul în care veți stabili marca datei și a orei în cadrul Windows.*

414 STABILIREA MĂRCII DATEI ȘI OREI UNUI FIȘIER LA DATA ȘI ORA CURENTE



Câteva dintre secțiunile acestei cărți arată modalități de marcare a datei și orei unui fișier. Atunci când doriți stabilirea mărcii datei și a orei unui fișier la data și ora curentă, puteți face aceasta foarte repede, cu funcția *utime*, cum arătăm în continuare:

```
#include <utime.h>

int utime(char *cale, struct utimbuf *data_ora);
```

Parametrul *cale* este un șir de caractere care specifică numele și directorul fișierului pe care îl doriți. Parametrul *data_ora* este o structură care conține data și ora la care fișierul a fost ultima oară modificat și accesat, cum arătăm în continuare:

```
struct utimbuf
   {
     time_t actime; // ultimul acces
     time_t modtime; // ultima modificare
   };
```

Dacă lucrați în mediul DOS, DOS utilizează numai data și ora modificării. Dacă invocați funcția *utime*, cu parametrul *data_ora* la valoarea *NULL*, funcția stabilește marca pentru dată și oră la data și ora curente. Dacă funcția se execută cu succes, va returna valoarea 0. Dacă apare o eroare, funcția va returna valoarea -1 și va stabili variabila globală *errno*. Următorul program, *utime.c*, utilizează funcția *utime* pentru a stabili marca datei și a orei unui fișier specificat, la data și ora curente:

```
#include <stdio.h>
#include <utime.h>

void main(int argc, char **argv)
{
   if (utime(argv[1], (struct utimbuf *) NULL))
     printf("Eroare la stabilirea datei si orei\n");
   else
     printf("Data si ora sunt stabilite\n");
}
```

Observație: *Secțiunea 1465 detaliază modul în care veți stabili marca datei și a orei în Windows.*

CITIREA ȘI SCRIEREA DATELOR CUVÂNT CU CUVÂNT

C/C++ 415

Așa cum ați învățat, funcțiile *getc și putc* vă permit scrierea și citirea informațiilor în fișiere octet cu octet. În funcție de conținutul fișierului dumneavoastră, puteți să scrieți și să citiți datele cuvânt cu cuvânt. Pentru a vă ajuta să realizați aceasta, majoritatea compilatoarelor de C dispun de funcțiile *getw* și *putw*, cum arătăm mai jos:

```
#include <stdio.h>

int getw(FILE *flux);
int putw(int cuvant, FILE *flux);
```

Dacă funcția *getw* se execută cu succes, ea va returna valoarea întreagă citită din fișier. Dacă apare o eroare, sau funcția *getw* întâlnește sfârșitul fișierului, ea va returna *EOF*. Dacă funcția *putw* se execută cu succes, va returna valoarea întreagă pe care a scris-o în fișier. Dacă apare o eroare, funcția *putw* va returna *EOF*. Următorul program, *putwgetw.c*, utilizează funcția *putw* pentru a scrie valorile de la 1 la 100 într-un fișier. Programul deschide apoi același fișier și citește valorile utilizând funcția *getw*, cum arătăm în continuare:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
```

```
{
  FILE *pointer_fisier;
  int cuvant;

  if ((pointer_fisier = fopen("DATA.DAT", "wb")) == NULL)
    {
     printf("Eroare la deschiderea DATA.DAT pentru iesire\n");
     exit(1);
    }
  else
    {
     for (cuvant = 1; cuvant <= 100; cuvant++)
       putw(cuvant, pointer_fisier);
     fclose(pointer_fisier);
    }

  if ((pointer_fisier = fopen("DATA.DAT", "rb")) == NULL)
    {
     printf("Eroare la deschiderea DATA.DAT pentru intrare\n");
     exit(1);
    }
  else
    {
     do
       {
         cuvant = getw(pointer_fisier);
         if ((cuvant == EOF) && (feof(pointer_fisier)))
           break;
         else
           printf("%d ", cuvant);
       }
     while (1);
     fclose(pointer_fisier);
    }
}
```

416 MODIFICAREA DIMENSIUNILOR UNUI FIȘIER



Atunci când lucrați cu fișiere, puteți fie să alocați o mare parte a spațiului discului pentru un fișier, fie să micșorați dimensiunile unui fișier. Pentru asemenea cazuri, programele dumneavoastră pot utiliza funcția *chsize*, cum se arată mai jos:

```
#include <io.h>

int chsize(int indicator, long dimensiune);
```

Parametrul *indicator* este indicatorul de fișier pe care funcțiile *open* sau *creat* l-a returnat anterior programului. Parametrul *dimensiune* specifică dimensiunea dorită a fișierului. Dacă funcția *chsize* se execută cu succes, ea va întoarce valoarea 0. Dacă apare o eroare, funcția

chsize va returna valoarea -1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 416.

Valoarea	Semnificație
EACCES	Acces nevalid
EBADF	Indicator de fișier nevalid
ENOSPC	Insuficient spațiu pe disc (Unix)

Tabelul 416 *Valorile de eroare returnate de funcția* ***chsize.***

Dacă măriți dimensiunile fișierului, atunci funcția *chsize* va umple noul spațiul al fișierului cu caractere *NULL*. Următorul program, *chsize.c*, creează un fișier numit *100zeros.dat* și apoi utilizează funcția *chsize* pentru a umple cu zerouri primii 100 de octeți ai fișierului:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

void main(void)
  {
    int indicator;

    if ((indicator = creat("100ZEROS.DAT", S_IWRITE)) == -1)
      fprintf(stderr, "Eroare la deschiderea 100ZEROS.DAT");
    else
      {
        if (chsize(indicator, 100L))
          printf("Eroare la modificarea dimensiunii fisierului\n");
        close(indicator);
      }
  }
```

CONTROLUL OPERAȚIILOR DE CITIRE ȘI SCRIERE CU FIȘIERE DESCHISE

C/C++417

Așa cum ați învățat, atunci când deschideți un fișier, dacă folosiți funcțiile *open, creat* sau *fopen*, trebuie să specificați dacă vreți să accesați fișierul în modul citire, scriere sau citire și scriere. Funcția *umask* permite fișierelor dumneavoastră să controleze modul în care programul va deschide mai târziu fișierele. Formatul funcției *umask* este următorul:

```
#include <io.h>

unsigned umask(unsigned mod_acces);
```

Parametrul *mod_acces* specifică modul în care doriți să preveniți utilizarea fișierelor. Valorile valide pentru parametrul *mod_acces* sunt arătate în tabelul 417.

Mod de acces	Semnificație
S_IWRITE	Previne accesul pentru scriere
S_IREAD	Previne accesul pentru citire
S_IWRTE\S_IREAD	Previne accesul pentru scriere și citire

Tabelul 417 *Valorile valide pentru parametrul* ***mod_acces*** *al funcției* ***umask.***

Ca un exemplu, dacă doriți să preveniți ca un program să deschidă fișierele cu modul de acces pentru scriere, atunci trebuie să folosiți funcția *umask* ca mai jos:

```
mod_vechi = umask(S_IWRITE);
```

Așa cum se arată în continuare, funcția returnează valoarea anterioară. Următorul program, *umask.c*, utilizează funcția *umask* pentru a stabili modul de acces la *S_IWRITE*, care va șterge bitul de acces la scriere al fișierului (făcând ca fișierul să fie *read-only*). Apoi programul creează și scrie ieșirea în fișierul *iesire.dat*. După ce programul închide fișierul, el încearcă să deschidă fișierul *iesire.dat* în modul de acces pentru scriere. Deoarece funcția *umask* a stabilit anterior fișierul ca fiind *read-only*, operațiunea de deschidere va eșua, cum arătăm în continuare:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

void main(void)
  {
    int iesire;
    int veche_setare;
    veche_setare = umask(S_IWRITE);
    if ((iesire = creat("IESIRE.DAT", S_IWRITE)) == -1)
      {
       fprintf(stderr, "Eroare la crearea IESIRE.DAT\n");
       exit(1);
      }
    else
      {
       if (write(iesire, "Test", 4) == -1)
         fprintf(stderr, "Nu se poate scrie in fisier\n");
       else
         printf("S-a reusit scrierea in fisier\n");
       close(iesire);
      }
    if ((iesire = open("IESIRE.DAT", O_WRONLY)) == -1)
      fprintf(stderr, "Eroare la deschiderea IESIRE.DAT
        pentru iesire\n");
    else
      printf("Fisierul a fost deschis pentru scriere\n");
  }
```

Observație: *Pentru a șterge fișierul **iesire.dat** de pe disc, trebuie să lansați comanda **ATTRIB-R iesire.dat** și apoi să ștergeți fișierul.*

ATRIBUIREA UNUI BUFFER DE FIȘIER

C/C++ 418

În capitolul despre tastatură al acestei cărți, ați învățat că limbajul C dispune de funcții I/O care execută intrări și ieșiri utilizând bufferul și directe. Pentru operațiile de I/O care utilizează bufferul, datele sunt scrise sau citite prin buffer, înainte de a fi disponibile pentru program. Operațiunile cu fișiere, de exemplu, utilizează I/O cu buffer. Atunci când programele dumneavoastră execută o operație I/O directă, pe de altă parte, datele sunt imediat disponibile programului, fără să mai fie plasate într-un buffer intermediar. Adesea puteți să folosiți operații I/O directe pentru a avea acces direct de la tastatură. De obicei, compilatorul de C alocă automat un buffer pentru fluxurile de fișier. Totuși, puteți utiliza funcția *setbuf* pentru a specifica propriul dumneavoastră buffer, cum arătăm în continuare:

```
#include <stdio.h>

void setbuf(FILE *flux, char *buffer);
```

Parametrul *flux* corespunde unui fișier deschis căruia doriți să-i atribuiți noul buffer. Parametrul *buffer* este un pointer la bufferul respectiv. Dacă parametrul *buffer* conține *NULL*, fișierul deschis pe care îl specifică *fluxul* nu va reține datele în buffer. Următorul program, *setbuf.c*, utilizează funcția *setbuf* pentru a modifica bufferul pe care compilatorul de C l-a atribuit indicatorului de fișier *stdout*. Programul scrie apoi o ieșire la *stdout*. Totuși, pentru că programul a plasat datele într-un buffer de mari dimensiuni, datele vor apărea pe ecran cu trei secunde de întârziere. Programul va umple apoi bufferul, caracter după caracter, la intervale de zece milisecunde între caractere. Când bufferul devine plin, acesta apare pe ecran, cum arătăm în continuare:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void main(void)
  {
    char buffer[512];
    int litera;

    setbuf(stdout, buffer);

    puts("Prima linie a iesirii");
    puts("A doua linie a iesirii");
    puts("A treia linie a iesirii");
    delay(3000);
    printf("Buferul este plin acum\n");
    fflush(stdout);
    for (litera = 0; litera < 513; litera++)
      {
        putchar('A');
        delay(10);
      }
  }
```

419 ALOCAREA UNUI BUFFER DE FIȘIER



În secțiunea 418 ați învățat cum se folosește funcția *setbuf* pentru a atribui un buffer unui fișier. Atunci când folosiți funcția *setbuf*, trebuie să specificați bufferul dorit. În mod similar, cele mai multe compilatoare de C dispun de funcția *setvbuf* care alocă un buffer (utilizând *malloc*) de dimensiunea dorită și apoi atribuie bufferul fișierului specificat. În plus, funcția *setvbuf* vă permite precizarea tipului dorit de buffer, cum se arată mai jos:

```
#include <stdio.h>

  int setvbuf(FILE *flux, char *buffer, int tip_buffer, size_t
    dim_buffer);
```

Parametrul *flux* este un pointer la un fișier deschis. Parametrul *buffer* este un pointer la bufferul în care compilatorul de C va păstra datele dumneavoastră. Dacă parametrul *buffer* este *NULL*, funcția *setvbuf* va aloca bufferul pentru dumneavoastră. Parametrul *tip_buffer* vă permite să controlați tipul bufferului. În sfârșit, parametrul *dim_buffer* vă permite să precizați dimensiunea bufferului până la 32767 de octeți. Dacă funcția *setvbuf* reușește execuția, ea va returna valoarea 0. Dacă apare o eroare (cum ar fi memorie insuficientă), funcția *setvbuf* va returna o valoare diferită de zero. Tabelul 419 listează valorile valide pentru parametrul *tip_buffer*.

Tip buffer	Păstrare în buffer
_IOFBF	Păstrează buffer plin. Când bufferul este gol, următoarea operație de citire va încerca să umple bufferul. Pentru ieșire, bufferul trebuie să fie plin, înainte ca funcția *setvbuf* să scrie datele pe disc.
_IOLBF	Păstrează linii. Când bufferul este gol, următoarea operație de citire va încerca să umple bufferul. Pentru ieșire, funcția *setvbuf* scrie bufferul pe disc atunci când bufferul este plin sau când funcția întâlnește caracterul de linie nouă.
_IONBF	Fără păstrare în buffer. Programul va executa operații I/O directe.

***Tabelul 419** Tipurile valide de buffer utilizate de funcția **setvbuf**.*

Următorul program, *setvbuf.c*, utilizează funcția *setvbuf* pentru a aloca un buffer de 8Kb pentru păstrarea în totalitate a fișierului în buffer:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void main(void)
  {
    char linie[512];
    char *buffer;
    FILE *intrare;

    if ((intrare = fopen("\\AUTOEXEC.BAT", "r")) == NULL)
      printf("Eroare la deschiderea \\AUTOEXEC.BAT\n");
    else
      {
        if (setvbuf(intrare, buffer, _IOFBF, 8192))
```

```
            printf("Eroare la modificarea bufferului\n");
        else
            while (fgets(linie, sizeof(linie), intrare))
              fputs(linie, stdout);
            fclose(intrare);
      }
  }
```

CREAREA UNUI NUME DE FIȘIER UNIC UTILIZÂND MKTEMP

C/C++ 420

Deoarece lucrați cu fișiere, capacitatea de a crea nume unice pentru fișierele temporare este foarte importantă. Unele dintre secțiunile acestui capitol au prezentat căi de creare a unor nume de fișiere aleatoare. În multe cazuri, veți dori să creați un nume unic de fișier, dar, de asemenea, veți dori ca numele de fișier să urmeze un anumit format pe care l-ați descris în cadrul aplicației. De exemplu, pentru un program de contabilitate, e posibil ca toate numele fișierelor dumneavoastră să înceapă cu literele *CONTAB*. Pentru a vă ajuta să controlați crearea numelui unic de fișier, multe compilatoare de C dispun de funcția *mktemp*, cum arătăm în continuare:

```
#include <dir.h>
char *mktemp(char *sablon);
```

Parametrul *sablon* este un pointer la un șir de caractere care conține șase caractere urmate de șase X-uri și de *NULL*. În exemplul dat mai sus, acest șablon ar trebui să fie un pointer la "CONTABXXXXXX". Funcția *mktemp* înlocuiește X-urile cu două caractere ale numelui de fișier, un punct și trei caractere pentru extensie. Dacă funcția *mktemp* reușește execuția, ea va returna un pointer la șirul șablon. Dacă apare o eroare, funcția va returna *NULL*. Deoarece funcția *mktemp* adaugă litere parametrului *sablon*, trebuie să vă asigurați că alocați 13 sau mai multe poziții pentru caractere în șir. Următorul program, *mktemp.c*, ilustrează modul de utilizare a funcției *mktemp*:

```
#include <stdio.h>
#include <dir.h>
void main(void)
  {
    char nume_a[13] = "CONTABXXXXXX";
    char nume_b[13] = "COMPUTXXXXXX";
    char nume_c[13] = "PCCHIPXXXXXX";
    if (mktemp(nume_a))
      puts(nume_a);
    if (mktemp(nume_b))
      puts(nume_b);
    if (mktemp(nume_c))
      puts(nume_c);
  }
```

Atunci când compilaţi şi executaţi programul *mktemp.c*, ecranul dumneavoastră va afişa următoarele:

```
CONTABAA.AAA
COMPUTAA.AAA
PCCHIPAA.AAA
C:\>
```

421 CITIREA ŞI SCRIEREA STRUCTURILOR



Capitolul despre structuri al acestei cărţi prezintă multe programe care lucrează cu structuri. Atunci când programele dumneavoastră lucrează cu structuri, adesea se vor ivi ocazii în care programele dumneavoastră vor trebui să păstreze structuri de date pe o dischetă sau pe hard-disc care vor fi citite mai târziu. Ca regulă, când trebuie să citiţi sau să scrieţi o structură, puteţi să trataţi structura ca pe un interval lung de octeţi. De exemplu, următorul program, *dtout.c*, utilizează funcţia *write* a limbajului C pentru a scrie data şi ora curente ale sistemului în fişierul *datetime.dat*:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>
#include <sys\stat.h>

void main(void)
  {
    struct date data_curenta;
    struct time ora_curenta;
    int handle;
    getdate(&data_curenta);
    gettime(&ora_curenta);
    if ((handle = creat("DATETIME.OUT", S_IWRITE)) == -1)
     fprintf(stderr, "Eroare la deschiderea fisierului
       DATETIME.OUT\n");
    else
     {
      write(handle, &data_curenta, sizeof(data_curenta));
      write(handle, &ora_curenta, sizeof(ora_curenta));
      close(handle);
     }
  }
```

După cum vedeţi, pentru a scrie structura, programul transmite, pur şi simplu, adresa structurii. În mod similar, următorul program, *dtin.c*, utilizează funcţia *read* pentru a citi structurile pentru dată şi oră:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>

void main(void)
```

```
{
  struct date data_curenta;
  struct time ora_curenta;
  int handle;

  if ((handle = open("DATETIME.OUT", O_RDONLY)) == -1)
    fprintf(stderr, "Eroare la deschiderea fisierului
      DATETIME.OUT\n");
  else
    {
      read(handle, &data_curenta, sizeof(data_curenta));
      read(handle, &ora_curenta, sizeof(ora_curenta));
      close(handle);
      printf("Data: %02d-%02d-%02d\n",
        data_curenta.da_mon, data_curenta.da_day,
        data_curenta.da_year);
      printf("Ora: %02d:%02d\n", ora_curenta.ti_hour,
        ora_curenta.ti_min);
    }
}
```

CITIREA DATELOR STRUCTURII DINTR-UN FLUX C/C++ 422

În secțiunea 421 ați învățat cum se utilizează funcțiile *read* și *write* ale limbajului C pentru a executa operații de I/O cu fișiere care utilizează structuri. Dacă programele dumneavoastră utilizează *fluxuri de fișiere*, în loc de indicatoare de fișier, pentru intrările și ieșirile de fișier puteți efectua același proces utilizând funcțiile *fread* și *fwrite*, cum se arată mai jos:

```
#include <stdio.h>

size_t fread(void *buffer, size_t dim_buffer,
  size_t nr_element, FILE *flux);
size_t fwrite(void *buffer, size_t dim_buffer,
  size_t nr_element, FILE *flux);
```

Parametrul *buffer* conține un pointer la datele pe care le doriți la ieșire. Parametrul *dim_buffer* specifică dimensiunea datelor în octeți. Parametrul *nr_element* specifică numărul de structuri pe care le scrieți și parametrul *flux* este un pointer la un *flux* de fișier deschis. Dacă funcția se execută cu succes, ea va returna numărul de elemente citite sau scrise. Dacă apare o eroare sau dacă funcțiile întâlnesc sfârșitul fișierului, ambele funcții vor returna valoarea 0. Următorul program, *dtoutf.c*, utilizează funcția *fwrite* pentru a scrie structurile dată și oră curente într-un fișier:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct date data_curenta;
```

```
    struct time ora_curenta;
    FILE *iesire;

    getdate(&data_curenta);
    gettime(&ora_curenta);
    if ((iesire = fopen("DATETIME.OUT", "w")) == NULL)
      fprintf(stderr, "Eroare la deschiderea fisierului
        DATETIME.OUT\n");
    else
      {
        fwrite(&data_curenta, sizeof(data_curenta), 1, iesire);
        fwrite(&ora_curenta, sizeof(ora_curenta), 1, iesire);
        fclose(iesire);
      }
  }
```

În mod asemănător, programul *dtinf.c* utilizează funcția *fread* pentru a citi valorile structurii, ca mai jos:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct date data_curenta;
    struct time ora_curenta;
    FILE *intrare;

    if ((intrare = fopen("DATETIME.OUT", "r")) == NULL)
      fprintf(stderr, "Eroare la deschiderea fisierului
        DATETIME.OUT\n");
    else
      {
        fread(&data_curenta, sizeof(data_curenta), 1, intrare);
        fread(&ora_curenta, sizeof(ora_curenta), 1, intrare);
        fclose(intrare);
        printf("Data: %02d-%02d-%02d\n",
          data_curenta.da_mon, data_curenta.da_day,
          data_curenta.da_year);
        printf("Ora: %02d:%02d\n", ora_curenta.ti_hour,
          ora_curenta.ti_min);
      }
  }
```

423 DUPLICAREA UNUI INDICATOR DE FIȘIER



Multe secțiuni din acest capitol prezintă funcții care lucrează cu indicatori de fișier. În programele dumneavoastră, puteți să duplicați valoarea unui indicator de fișier. De exemplu, dacă programul dumneavoastră execută operații de I/O critice, puteți să duplicați un

indicator de fișier și apoi să închideți noul indicator copiat pentru a salva conținutul fișierului pe disc. Deoarece primul indicator rămâne deschis, nu trebuie să redeschideți fișierul după operația de salvare, cum prezentăm mai jos:

```
#include <io.h>

int dup(int indicator);
```

Parametrul *indicator* este indicatorul fișierului deschis pe care doriți să-l duplicați. Dacă funcția *dup* reușește duplicarea indicatorului, ea va returna o valoare pozitivă. Dacă apare o eroare, funcția *dup* va returna -1. Următorul program, *dup.c*, ilustrează modul de utilizare a funcției *dup* pentru salvarea bufferelor fișierului pe disc:

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>

void main(void)
  {
    int indicator;
    int indicator_copiat;
    char titlu[] = "Jamsa\'s C/C++ Programmer\'s Bible!";
    char sectiune[] = "Fisiere";

    if ((indicator = open("IESIRE.TST", O_WRONLY | O_CREAT,
    S_IWRITE)) == -1)
      printf("Eroare la deschierea IESIRE.TST\n");
    else
      {
        if ((indicator_copiat = dup(indicator)) == -1)
          printf("Eroare la duplicarea indicatorului\n");
        else
          {
            write(indicator, titlu, sizeof(titlu));
            close(indicator_copiat); // Scrie bufferul
            write(indicator, sectiune, sizeof(sectiune));
            close(indicator);
          }
      }
  }
```

FORȚAREA VALORII UNUI INDICATOR DE FIȘIER C/C++ 424

În secțiunea 423 ați învățat cum să folosiți comanda *dup* pentru a realiza o copie (duplicat) a conținutului unui indicator de fișier. Puteți să modificați valoarea unui indicator de fișier deschis și să-i atribuiți valoarea unui indicator diferit. Când executați modificarea și stabiliți operațiile cu fișiere, puteți utiliza funcția *dup2*, ca mai jos:

```
#include <io.h>

int dup2(int indicator_sursa, int indicator_destinatie);
```

Parametrul *indicator_destinatie* este indicatorul de fişier a cărui valoare doriţi să o actualizaţi. Dacă funcţia *dup2* reuşeşte să atribuie indicatorul, ea va returna valoarea 0. Dacă apare o eroare, funcţia va returna -1. Parametrul *indicator_sursa* este indicatorul de fişier a cărui valoare doriţi să o atribuiţi destinaţiei. Următorul program, *dup2.c*, utilizează funcţia *dup2* pentru a atribui valoarea funcţiei *stderr* la *stdout*. Astfel, utilizatorii nu pot redirecta ieşirea programului de la ecranul calculatorului:

```
#include <stdio.h>
#include <io.h>

void main(void)
  {
    dup2(2, 1); // stdout are indicator 1, stderr are indicator 2
    printf("Acest mesaj nu poate fi redirectat!\n");
  }
```

425 ASOCIEREA UNUI INDICATOR DE FIŞIER CU UN FLUX

C/C++

Multe secţiuni din acest capitol prezintă funcţii care lucrează fie cu fluxuri, fie cu indicatoare de fişier. În programul dumneavoastră, atunci când operaţi cu indicatori de fişier, puteţi să utilizaţi o funcţie care corespunde unui flux. În astfel de cazuri, programul dumneavoastră poate utiliza funcţia *fdopen* pentru a asocia un indicator de fişier cu un flux, ca mai jos:

```
#include <stdio.h>

FILE *fdopen(int indicator, char *mod_acces);
```

Parametrul *indicator* este indicatorul unui fişier deschis, pe care doriţi să-l asociaţi cu un flux. Parametrul *mod_acces* este un pointer la un şir de caractere care specifică modul în care doriţi să utilizaţi fluxul. Valoarea parametrului *mod_acces* trebuie să fie una dintre valorile modului de acces pe care de obicei îl folosiţi cu funcţia *fopen*. Dacă funcţia reuşeşte, ea returnează un pointer la flux. Dacă apare o eroare, funcţia returnează *NULL*. Următoarea instrucţiune, de exemplu, asociază indicatorul *intrare* cu pointerul la fişier *fpointer* pentru acces la scriere:

```
if ((fpointer = fdopen(intrare, "r")) == NULL)
  printf("Eroare la asocierea fisierului\n");
else
  {
    gets(sir, sizeof(sir), fpointer);
    fclose(fpointer);
  }
```

PARTAJAREA FIŞIERULUI

C/C++426

Dacă operaţi într-un mediu de reţea şi aţi instalat comanda DOS SHARE, puteţi scrie programe care permit ca mai multe programe să acceseze simultan părţi diferite ale aceluiaşi fişier. De exemplu, să considerăm un program care permite mai multor utilizatori să repartizeze locuri într-un avion. Când un utilizator vrea să repartizeze un anumit loc, programul blochează acel loc, astfel încât, un alt utilizator nu îl va repartiza. După ce programul repartizează locul, utilizatorul deblochează locul.

Atunci când partajaţi fişiere în acest mod, trebuie mai întâi să utilizaţi funcţia *sopen* pentru a deschide fişierul pentru partajare. Apoi, când programul dumneavoastră doreşte să acceseze un interval de octeţi din fişier, programul încearcă să blocheze datele. Dacă nimeni altcineva nu utilizează (blochează) datele în acel moment, atunci blocajul din program reuşeşte. După ce programul încheie operaţiunile cu datele, el poate debloca intervalul de octeţi din fişier.

Când un program blochează un interval de octeţi dintr-un fişier, programul poate atribui un blocaj care va permite altor utilizatori să acceseze datele în anumite modalităţi. De exemplu, programul poate permite altui fişier să citească intervalul blocat sau poate permite altui program să scrie şi să citească acelaşi interval de octeţi. Multe din secţiunile care urmează prezintă funcţii din biblioteca *run-time* de C, care acceptă partajarea şi blocarea fişierelor.

DESCHIDEREA UNUI FIŞIER PENTRU ACCES PARTAJAT

C/C++427

În secţiunea 426 aţi învăţat că puteţi folosi comanda DOS SHARE pentru a deschide fişiere pentru ca mai multe programe să le utilizeze în acelaşi timp. Pentru a deschie un fişier pentru utilizare partajată, programul dumneavoastră trebuie să folosească funcţia *sopen*, al cărei prototip îl prezentăm în continuare:

```
#include <share.h>
int sopen(char *nume_cale, int mod_acces,
  int semn_partaj[, int mod_creare]);
```

Parametrii *nume_cale, mod_acces şi mod_creare* sunt similari cu cei utilizaţi de funcţia *open*. Parametrul *semn_partaj* specifică modul în care programe diferite pot partaja fişierul. Dacă funcţia *sopen* reuşeşte să deschidă fişierul, ea va returna un indicator de fişier. Dacă apare o eroare, funcţia *sopen* va returna -1. Tabelul 427 listează valorile valide ale parametrului *semn_partaj*:

Semn partajare	Partajare permisă
SH_COMPAT	Permite partajare compatibilă
SH_DENYRW	Previne accesul la scriere şi citire
SH_DENYWR	Previne accesul la scriere
SH_DENYRD	Previne accesul la citire
SH_DENYNONE	Permite orice acces (citire şi scriere)
SH_DENYNO	Permite orice acces (citire şi scriere)

***Tabelul 427** Modurile de acces partajat acceptate de funcţia **sopen**.*

Următorul program, *sopen.c*, deschide fişierul specificat în linia de comandă pentru acces partajat la citire. Fişierul aşteaptă apoi să apăsaţi o tastă înaintea citirii şi afişării conţinutului din fişier, ca mai jos:

```
#include <stdio.h>
#include <share.h>
#include <io.h>
#include <fcntl.h>

void main(int argc, char *argv[])
  {
    int indicator, octeti_cititi;
    char buffer[256];

    if ((indicator = sopen(argv[1], O_RDONLY, SH_DENYWR)) == -1)
      printf("Eroare la deschiderea fisierului %s\n", argv[1]);
    else
      {
        printf("Apasati Enter pentru continuare\n");
        getchar();
        while (octeti_cititi = read(indicator, buffer,
           sizeof(buffer)))
          write(1, buffer, octeti_cititi); // 1 este stdout
        close(indicator);
      }
  }
```

Pentru a înţelege mai bine cum operează programul *sopen.c* daţi comanda SHARE. Apoi, porniţi Windows şi creaţi o fereastră DOS în care rulaţi programul folosind numele de fişier *sopen.c* ca fişier partajat. Când programul vă solicită să apăsaţi o tastă, deschideţi o a doua fereastră DOS şi folosiţi TYPE pentru afişarea conţinutului fişierului. Întrucât TYPE afişează conţinutul fişierului *sopen.c*, două programe au acelaşi fişier deschis în acelaşi timp. Închideţi fereastra şi reveniţi la prima fereastră. Apăsaţi Enter pentru a afişa conţinutul fişierului. Experimentaţi în continuare programul *sopen.c* trecând prin modurile de partajare. Repetaţi procesul de accesare a fişierului folosind două programe.

428 BLOCAREA CONŢINUTULUI UNUI FIŞIER



Aşa cum aţi învăţat, atunci când partajaţi conţinutul unui fişier, este posibil să blocaţi un interval de octeţi în cadrul fişierului pentru a preveni ca un alt program să îl modifice. Pentru a bloca un anumit interval de octeţi dintr-un fişier, programele dumneavoastră pot utiliza funcţia *lock*, arătată mai jos:

```
#include <io.h>

int lock(int indicator, long poz_start, long nr_octeti);
```

Parametrul *indicator* este un indicator care corespunde unui fişier pe care funcţia *sopen* l-a deschis pentru partajare. Parametrul *poz_start* specifică deplasamentul de la începutul intervalului de octeţi din cadrul fişierului, pe care doriţi să îl blocaţi. Parametrul *nr_octeti* specifică numărul de octeţi pe care doriţi să îi blocaţi. Dacă funcţia *lock* reuşeşte să blocheze

intervalul de octeți, ea va returna valoarea 0. Dacă apare o eroare, funcția va returna valoarea -1. Pentru ca funcția *lock* să se execute, trebuie să aveți instalată comanda DOS SHARE.

După ce blocați intervalul de octeți, alte programe vor încerca de trei ori să citească sau să scrie intervalul blocat. După cea de a treia încercare nereușită a programului de a citi datele, funcțiile *read* sau *write* vor returna eroare. Următorul program, *lockauto.c*, blochează mai întâi primii cinci octeți ai fișierului *autoexec.bat* din directorul rădăcină și apoi așteaptă să fie apăsată o tastă:

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>

void main(void)
  {
    int indicator;

    if ((indicator = sopen("\\AUTOEXEC.BAT", O_RDONLY,
      SH_DENYNO)) == -1)
     printf("Eroare la deschiderea AUTOEXEC.BAT\n");
    else
      {
       lock(indicator, 0L, 5L);
       printf("Fisier blocat--apasati Enter pentru continuare\n");
       getchar();
       close(indicator);
      }
  }
```

Apoi, următorul program, *incerc.c*, încearcă să citească fișierul *autoexec.bat* octet cu octet. Dacă apare o eroare în timpul citirii fișierului, programul va afișa un mesaj de eroare, cum se arată mai jos:

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>

void main(void)
  {
    int indicator;
    int deplasament = 0;
    int octeti_cititi;
    char buffer[128];

    if ((indicator = sopen("\\AUTOEXEC.BAT",
      O_BINARY | O_RDONLY, SH_DENYNO)) == -1)
     printf("Eroare la deschiderea AUTOEXEC.BAT\n");
    else
      {
```

```
      while (octeti_cititi = read(indicator, buffer, 1))
       {
         if (octeti_cititi == -1)
          printf("Eroare la citirea deplasamentului %d\n",
             deplasament);
         else
          write(1, buffer, octeti_cititi);
         deplasament++;
         lseek(indicator, deplasament, SEEK_SET);
       }
      close(indicator);
    }
 }
```

429 UN CONTROL MAI BUN AL BLOCĂRII FIȘIERELOR

C/C++

În secțiunea 428 ați învățat să folosiți funcția *lock* pentru a bloca un interval de octeți dintr-un fișier. Atunci când folosiți funcția *lock*, operația, fie reușește, fie eșuează imediat. Dacă doriți un mai bun control asupra operației de blocare, puteți utiliza funcția *locking*, cum arătăm în continuare:

```
#include <io.h>
#include <sys\locking.h>

int locking(int indicator, int comanda_bloc, long nr_octeti);
```

Parametrul *indicator* este un indicator asociat cu fișierul pe care vreți să îl blocați. Parametrul *comanda_bloc* specifică operația de blocare dorită. Parametrul *nr_octeti* specifică numărul de octeți pe care doriți să îi blocați. Începutul zonei de blocare depinde de poziția curentă a pointerului de fișier. Dacă doriți să blocați o anumită zonă, puteți să utilizați mai întâi funcția *lseek* pentru a poziționa pointerul de fișier. Tabelul 429.1 specifică valorile posibile ale parametrului *comanda_bloc*.

Comanda de blocare	Semnificație
LK_LOCK	Blochează zona specificată. Dacă blocajul nu are loc, funcția *locking* va încerca o dată la fiecare secundă, timp de 10 secunde, să aplice blocajul.
LK_RLCK	Execută aceeași operație ca *LK_LOCK*
LK_NBLCK	Blochează zona specificată. Dacă blocajul nu are loc, funcția *locking* va returna imediat eroare.
LK_UNLCK	Deblochează o zonă blocată anterior.

***Tabelul 429.1** Comenzile utilizate de funcția **locking**.*

Dacă funcția *locking* reușește blocarea fișierului, ea va returna valoarea 0. Dacă apare o eroare, funcția *locking* va returna valoarea -1 și va stabili variabila globală *errno* la una dintre valorile specificate în tabelul 429.2.

Eroarea	Semnificație
EBADF	Indicator de fișier nevalid
EACCES	Fișier deja blocat sau neblocat
EDEADLOCK	Fișierul nu poate fi blocat după 10 încercări
EINVAL	Comanda specificată este nevalidă

Tabelul 429.2 *Valorile de eroare returnate de funcția* ***locking.***

Următorul program, *locking.c*, modifică programul *lockauto.c*, prezentat în secțiunea 428, pentru a utiliza funcția *locking* în scopul blocării primilor cinci octeți din *autoexec.bat*:

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>
#include <sys\locking.h>

void main(void)
  {
   int indicator;

   if ((indicator = sopen("\\AUTOEXEC.BAT", O_RDONLY,
     SH_DENYNO)) == -1)
     printf("Eroare la deschiderea AUTOEXEC.BAT\n");
   else
     {
       printf("Incearca sa blocheze fisierul\n");
       if (locking(indicator, LK_LOCK, 5L))
         printf("Eroare la blocarea fisierului\n");
       else
         {
           printf("Fisier blocat--apasati Enter pentru
             continuare\n");
           getchar();
           close(indicator);
         }
     }
  }
```

La fel ca mai sus, dacă aveți instalat Windows, încercați să rulați programul *locking.c* cu două ferestre DOS, în același timp.

Observație: *Înainte de a putea utiliza funcția* ***locking****, trebuie să instalați comanda DOS SHARE.*

LUCRUL CU DIRECTOARELE DOS

C/C++430

În cadrul programelor dumneavoastră în C, puteți să folosiți funcțiile *findfirst* și *findnext* pentru a lucra cu fișiere care se potrivesc cu o anumită combinație de caractere de înlocuire (de exemplu *"*.exe"*). Deoarece DOS nu tratează directoarele ca fișiere, programele dumneavoastră

nu pot utiliza serviciile DOS pentru a deschide un director și a-i citi conținutul. Dacă însă înțelegeți cum pune DOS informațiile pe disc, programele dumneavoastră pot citi din tabela de alocare a fișierelor și din directorul rădăcină și apoi pot citi urmărind sectoarele care conțin intrările unui director. Comenzile utilitare de disc (cum ar fi UNDELETE) și instrumentul de sortare a directorului execută aceste operații de I/O la nivel inferior, pe disc. Multe din următoarele secțiuni vor ilustra modalitățile în care programele dumneavoastră pot utiliza aceste funcții de I/O proprii directoarelor. Pentru a simplifica operația de citire a unui director, unele compilatoare de C dispun de funcțiile prezentate în tabelul 430.

Funcția	**Utilizare**
closedir	Închide fluxul unui director
opendir	Deschide fluxul unui director pentru operații de citire
readdir	Citește următoarea intrare în fluxul directorului
rewinddir	Mută pointerul în fluxul directorului, înapoi la începutul listei directorului

***Tabelul 430** Funcțiile de I/O cu directoare și utilizările lor.*

431 DESCHIDEREA UNUI DIRECTOR



În secțiunea 430 ați învățat că multe compilatoare de C dispun de funcții care vă permit să deschideți și să citiți numele fișierelor care există într-un anumit director. Pentru a deschide un director pentru operații de citire, programele dumneavoastră pot utiliza funcția *opendir*, ca mai jos:

```
#include <dirent.h>
DIR *opendir(char *nume_director);
```

Parametrul *nume_director* este un pointer la un șir de caractere care conține numele directorului dorit. Dacă numele directorului este *NULL*, funcția *opendir* deschide directorul curent. Dacă funcția *opendir* reușește, ea returnează un pointer la o structură de tip DIR. Dacă apare o eroare, funcția returnează *NULL*. Următoarea instrucțiune, de exemplu, ilustrează cum deschideți directorul DOS pentru operații de citire:

```
struct DIR *director_intrare;
if ((director_intrare = opendir("\\DOS")) == NULL)
   printf("Eroare la deschiderea directorului\n");
else
   // instructiuni
```

După ce ați executat operațiile de citire a directorului, va trebui să închideți fluxul directorului, utilizând funcția *closedir*, ca mai jos:

```
#include <dirent.h>
void closedir(DIR *director);
```

CITIREA UNEI INTRĂRI DIN DIRECTOR

C/C++ 432

În secțiunea 431 ați învățat cum se utilizează funcția *opendir* pentru a deschide lista unui director. După ce ați deschis directorul, puteți să utilizați funcția *readdir* pentru a citi numele următoarei intrări în lista directorului, ca mai jos:

```
#include <dirent.h>
struct dirent readdir(DIR *pointer_director);
```

Parametrul *pointer_director* este un pointer pe care funcția *opendir* îl returnează. Dacă funcția *readdir* reușește să citească intrarea din director, ea va returna intrarea citită. Dacă apare o eroare sau funcția *readdir* ajunge la sfârșitul directorului, funcția va returna *NULL*. Funcția *readdir* citește toate intrările din lista directorului, inclusiv intrările "." și "..".

UTILIZAREA SERVICIILOR PENTRU DIRECTOARE LA CITIREA C:\WINDOWS

C/C++ 433

În secțiunea 431, ați învățat cum se deschide și se închide lista unui director. În secțiunea 432, ați învățat cum se folosește funcția *readdir* pentru a citi următoarea intrare în lista directorului. Următorul program, *vezidir.c*, utilizează intrările din directorul bibliotecii *run-time* pentru a deschide, a citi și apoi a închide directorul specificat în linia de comandă:

```
#include <stdio.h>
#include <dirent.h>

void main(int argc, char *argv[])
  {
    DIR *pointer_director;
    struct dirent *intrare;

    if ((pointer_director = opendir(argv[1])) == NULL)
      printf("Eroare la deschiderea %s\n", argv[1]);
    else
      {
        while (intrare = readdir(pointer_director))
          printf("%s\n", intrare);
        closedir(pointer_director);
      }
  }
```

Următoarea comandă, de exemplu, utilizează programul *vezidir.c* pentru a afișa numele fișierelor din directorul *c:\windows*:

```
C:\> VEZIDIR C:\WINDOWS <ENTER>
```

REDESFĂȘURAREA UNUI DIRECTOR

C/C++ 434

În secțiunea 432, ați învățat că limbajul C vă pune la dispoziție funcții de bibliotecă *run-time* care vă permit să deschideți și să citiți numele fișierelor dintr-un anumit director. Atunci când citiți directoare, puteți să reîncepeți citirea fișierelor de la începutul listei directorului. O cale pentru a executa această operație este închiderea și apoi redeschiderea listei directorului. (

altă modalitate pe care o pot utiliza programele dumneavoastră este apelarea funcției *rewinddir*, ca mai jos:

```
#include <dirent.h>

void rewinddir(DIR *pointer_director);
```

Parametrul *pointer_director* este un pointer la lista directorului pe care doriți să o revedeți. Dacă doriți să încercați utilizarea funcției *rewinddir*, veți găsi că este mult mai rapidă decât închiderea și redeschiderea listei directorului.

435 CITIREA RECURSIVĂ A FIȘIERELOR DISCULUI



În secțiunea 433 ați utilizat programul *vezidir.c* pentru a afișa fișierele din lista unui director. Următorul program, *fisiere.c*, utilizează funcțiile de bibliotecă *run-time* pentru a afișa numele fiecărui fișier de pe discul dumneavoastră. Pentru a face aceasta, programul utilizează funcția recursivă *arata_dir*, ca mai jos:

```
#include <stdio.h>
#include <dirent.h>
#include <dos.h>
#include <io.h>
#include <direct.h>
#include <string.h>

void arata_dir(char *nume_director)
  {
    DIR *pointer_director;
    struct dirent *intrare;
    unsigned atribute;

    if ((pointer_director = opendir(nume_director)) == NULL)
     printf("Eroare la deschiderea %s\n", nume_director);
    else
     {
       chdir(nume_director);
       while (intrare = readdir(pointer_director))
        {
         atribute = _chmod(intrare, 0);
           // verifica daca intrarea e pentru un subdirector
           //si nu "." sau ".."
         if ((atribute & FA_DIREC) &&
          (strncmp(intrare, ".", 1) != 0))
          {
           printf("\n\n----%s----\n", intrare);
           arata_dir(intrare);
          }
         else
           printf("%s\n", intrare);
        }
```

```
        closedir(pointer_director);
        chdir("..");
      }
  }
void main(void)
  {
    char buffer[MAXPATH];
    // Salveaza directorul curent pentru utilizare ulterioara
    getcwd(buffer, sizeof(buffer));
    arata_dir ("\\");
    chdir(buffer);
  }
```

DETERMINAREA POZIŢIEI CURENTE ÎN FIŞIER

C/C++ 436

Aţi învăţat anterior cum urmăreşte compilatorul de C poziţia curentă în fişierele deschise pentru operaţii de intrare sau ieşire. În programele dumneavoastră, puteţi să determinaţi valoarea poziţiei pointerului. Dacă lucraţi cu fluxuri, puteţi să utilizaţi funcţia *ftell* pentru a determina poziţia pointerului de fişier. Dacă lucraţi cu indicatori fişier, însă, programele dumneavoastră pot utiliza funcţia *tell*, ca mai jos:

```
#include <stdio.h>

long tell(int indicator);
```

Funcţia *tell* returnează o valoare de tip *long* care specifică octetul deplasament al poziţiei curente în fişierul specificat. Următorul program, *tell.c*, utilizează funcţia *tell* pentru a afişa informaţii despre poziţia pointerului. Programul începe cu deschiderea fişierului din directorul rădăcină *config.sys*, în modul pentru citire. Programul utilizează apoi funcţia *tell* pentru a afişa poziţia curentă. În continuare, programul citeşte şi afişează conţinutul fişierului. După ce programul întâlneşte sfârşitul fişierului, el utilizează din nou funcţia *tell* pentru a afişa poziţia curentă, ca mai jos:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
void main(void)
  {
    int indicator;
    char buffer[512];
    int octeti_cititi;
    if ((indicator = open("\\CONFIG.SYS", O_RDONLY)) == -1)
      printf("Eroare la deschierea \\CONFIG.SYS\n");
    else
      {
        printf("Pozitia curenta in fisier %ld\n", tell(indicator));
```

```
      while (octeti_cititi = read(indicator, buffer,
         sizeof(buffer)))
        write(1, buffer, octeti_cititi);
      printf("Pozitia curenta in fisier %ld\n",
         tell(indicator));
      close(indicator);
    }
}
```

437 DESCHIDEREA UNUI FLUX PARTAJAT DE FIȘIER



Câteva dintre secțiunile acestui capitol prezintă modalități de partajare și de blocare a fișierelor, utilizând indicatorii fișier. Dacă de obicei lucrați cu fluxuri, programele dumneavoastră pot utiliza funcția *_fsopen*, cum se arată în continuare:

```
#include <stdio.h>
#include <share.h>

FILE * _fsopen(const char *nume_fisier, const *mod_acces,
   int semn_partaj);
```

Parametrii *nume_fisier* și *mod_acces* conțin pointeri șir de caractere la numele fișierului dorit și la modul de acces pe care în mod normal îl utilizează funcția *fopen*. Parametrul *semn_partaj* specifică modul de partajare. Dacă funcția reușește execuția, ea va returna un pointer la fișier. Dacă apare o eroare, funcția va returna *NULL*. Tabelul 437 listează valorile valide pe care puteți să le atribuiți parametrului *semn_partaj*.

Semn partajare	Partajare permisă
SH_COMPAT	Permite partajare compatibilă
SH_DENYRW	Previne accesul la scriere și citire
SH_DENYWR	Previne accesul la scriere
SH_DENYRD	Previne accesul la citire
SH_DENYNONE	Permite orice acces (citire și scriere)
SH_DENYNO	Permite orice acces (citire și scriere)

Tabelul 437 *Valorile valide pentru parametrul* ***semn_partaj.***

Următoarele instrucțiuni, de exemplu, deschid fișierul *autoexec.bat* din directorul rădăcină pentru operații de citire partajată:

```
if ((pointer_fisier = _fsopen("\\AUTOEXEC.BAT", "r",
SH_DENYWR)) == NULL)
    printf("Eroare la deschiderea \\AUTOEXEC.BAT\n");
else
    // instructiuni
```

CREAREA UNUI FIȘIER UNIC ÎNTR-UN ANUMIT DIRECTOR

C/C++438

Câteva dintre secțiunile acestui capitol au prezentat modalități prin care programele dumneavoastră pot crea fișiere temporare. Dacă lucrați în mod obișnuit cu indicatoare de fișier, puteți utiliza funcția *creattemp* care returnează un indicator, cum arătăm în continuare:

```
#include <dos.h>

int creattemp(char *cale, int atribut);
```

Parametrul *cale* specifică numele directorului în cadrul căruia doriți să creați fișierul. Numele trebuie să se termine prin două caractere *backslash* ('\\'). Funcția *creattemp* va adăuga numele fișierului la șirul de caractere pentru a produce numele de cale complet. Parametrul *atribut* specifică atributele fișierului dorit (sau 0 pentru nici unul). Tabelul 438 listează valorile valide pentru parametrul *atribut*.

Constanta	Descrierea
FA_RDONLY	Fișier read-only
FA_HIDDEN	Fișier ascuns
FA_SYSTEM	Fișier sistem

Tabelul 438 *Valorile valide pentru parametrul* ***atribut.***

Dacă funcția reușește execuția, ea va returna un indicator de fișier. Dacă apare o eroare, funcția va returna -1. Următorul program, *creattmp.c*, utilizează funcția *creattemp* pentru a crea un fișier unic în directorul TEMP:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>

void main(void)
  {
    char cale[64] = "C:\\TEMP\\";
    int indicator;

    if ((indicator = creattemp(cale, 0)) == -1)
      printf("Eroare la crearea fisierului\n");
    else
      {
        printf("Calea completa: %s\n", cale);
        close(indicator);
      }
  }
```

CREAREA UNUI FIȘIER NOU

C/C++439

Câteva dintre secțiunile acestui capitol au prezentat modalități de creare a fișierelor. În multe cazuri, dacă încercați să creați un fișier și numele specificat în funcție există deja, funcția va trunchia conținutul fișierului. Însă, de obicei creați un fișier dacă nu există altul cu același

nume. Pentru asemenea cazuri, programele dumneavoastră pot utiliza funcția *creatnew*, ca mai jos:

```
#include <dos.h>

int creatnew(const char *numecale, int atribut);
```

Parametrul *numecale* specifică numele complet de cale al fișierului pe care doriți să îl creați. Parametrul *atribut* specifică atributele fișierului dorit (sau 0 pentru nici unul). Tabelul 439.1 listează valorile posibile pentru parametrul *atribut*.

Atribut	Semnificație
FA_RDONLY	Fișier read-only
FA_HIDDEN	Fișier ascuns
FA_SYSTEM	Fișier sistem

Tabelul 439.1 *Valorile posibile pentru parametrul* ***atribut*** *al funcției* ***creatnew.***

Dacă funcția *creatnew* reușește execuția, ea va returna un indicator de fișier. Dacă apare o eroare, funcția va returna valoarea -1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 439.2.

Eroarea	Semnificația
EXISTS	Fișierul deja există
ENOENT	Calea nu este găsită
EMFILE	Prea multe fișiere deschise
EACCES	Violare de acces

Tabelul 439.2 *Valorile de eroare returnate de funcția* ***creatnew.***

Următorul program, *creatnew.c*, utilizează funcția *creatnew* pentru a crea un fișier numit *nou.dat* în directorul curent. Rulați acest program și încercați să creați fișierul de mai multe ori, cum arătăm în continuare:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>

void main(void)
  {
    int indicator;

    if ((indicator = creatnew("NOU.DAT", 0)) == -1)
     printf("Eroare la crearea NOU.DAT\n");
    else
      {
       printf("Fisier creat cu succes\n");
       close(indicator);
      }
  }
```

UTILIZAREA SERVICIILOR DOS PENTRU ACCESAREA FIȘIERELOR

C/C++ 44

Așa cum ați învățat, atunci când programele dumneavoastră accesează mai mult de 20 fișiere, puteți utiliza serviciile DOS, care vă permit să evitați rutinele bibliotecii run-time limbajului C. Următorul program, *copiere.c*, utilizează serviciile DOS pentru a cop conținutul primului fișier specificat în linia de comandă în cel de al doilea:

```
#include <stdio.h>
#include <dos.h>
void main(int argc, char **argv)
  {
    union REGS inregs, outregs;
    struct SREGS segs;
    char buffer[256];
    unsigned indicator_sursa, indicator_destinatie;
    if (*argv[1] && *argv[2])
      {
        // Deschide fisierul pentru copiere
        inregs.h.ah = 0x3D;
        inregs.h.al = 0; // Deschide fisierul pentru acces la citir
        inregs.x.dx = (unsigned) argv[1];
        segread (&segs);
        intdosx(&inregs, &outregs, &segs);
        if (outregs.x.cflag)
          printf ("Eroare la deschiderea fisierului sursa %s\n",
            argv[1]);
        else
          {
            indicator_sursa = outregs.x.ax;
            // Creaza fisierul destinatie, trunchind un
            // fisier existent cu acelasi nume
            inregs.h.ah = 0x3C;
            inregs.x.cx = 0; // Deschide cu atribut normal
            inregs.x.dx = (unsigned) argv[2];
            intdosx (&inregs, &outregs, &segs);
            if (outregs.x.cflag)
              printf ("Eroare la crearea fisierului destinatie
                %s\n", argv[2]);
            else
              {
                indicator_destinatie = outregs.x.ax;
                do {
                  // Citeste datele sursa
                  inregs.h.ah = 0x3F;
                  inregs.x.bx = indicator_sursa;
```

```
            inregs.x.cx = sizeof(buffer);
            inregs.x.dx = (unsigned) buffer;
            intdosx (&inregs, &outregs, &segs);
            if (outregs.x.cflag)
              {
                printf ("Eroare la citirea fisierului sursa\n");
                break;
              }
            else if (outregs.x.ax) // Nu e sfarsitul fisierului
              {
                // Scrie datele
                inregs.h.ah = 0x40;
                inregs.x.bx = indicator_destinatie;
                inregs.x.cx = outregs.x.ax;
                inregs.x.dx = (unsigned) buffer;
                intdosx (&inregs, &outregs, &segs);
                if (outregs.x.cflag)
                  {
                     printf ("Eroare la scrierea fisierului
                       destinatie\n");
                     break;
                  }
              }
          } while (outregs.x.ax != 0);
          // Inchide fisierele
          inregs.h.ah = 0x3E;
          inregs.x.bx = indicator_sursa;
          intdos (&inregs, &outregs);
          inregs.x.bx = indicator_destinatie;
          intdos (&inregs, &outregs);
       }
     }
   }
  else
    printf ("Specifica numele fisierelor sursa si destinatie\n");
}
```

Observație: *Așa cum ați învățat, veți utiliza, în general, interfața API Windows pentru a executa activități echivalente cu servicii DOS, sub Windows. Secțiunea 1471 explică modul în care se copiază fișierele utilizând Windows API.*

441 FORȚAREA DESCHIDERII UNUI FIȘIER ÎN MOD BINAR SAU TEXT



Ați învățat anterior că multe dintre compilatoarele de C utilizează variabila globală *_fmode* pentru a determina dacă un program a fost deschis în mod text sau binar. Atunci când folosiți funcția *fopen*, puteți să controlați modul pe care funcția *fopen* îl utilizează prin plasarea literei *t* sau *b* imediat după modul dorit, ca în tabelul 441.

Specificatorul de acces	Modul de acces
ab	Acces pentru adăugare în mod binar
at	Acces pentru adăugare în mod text
rb	Acces pentru citire în mod binar
rt	Acces pentru citire în mod text
wb	Acces pentru scriere în mod binar
wt	Acces pentru scriere în mod text

Tabelul 441 *Specificatorii modului de acces al fişierelor pentru funcţia **fopen**.*

Următoarea instrucţiune *fopen*, de exemplu, deschide fişierul *numefis.ext* pentru acces la citire în mod binar:

```
if ((pointer_fisier = fopen("NUMEFIS.EXT", "rb")))
```

CITIREA LINIILOR DE TEXT

C/C++ 442

Când programele dumneavoastră citesc fişiere text, citirea se face rând cu rând. Pentru a citi o linie dintr-un fişier, programul poate utiliza funcţia *fgets*, al cărei format este următorul:

```
#include <stdio.h>

char *fgets(char sir, int limita, FILE *flux);
```

Parametrul *sir* este bufferul de caractere în care *fgets* citeşte datele din fişiere. De regulă programele dumneavoastră vor declara o matrice de 128 sau 256 de octeţi pentru a reţine datele. Parametrul *limita* precizează numărul de caractere pe care le reţine bufferul. Când *fgets* citeşte caracterele din fişier, *fgets* va citi fie până la *limita*-1 (*limita* minus unu), fie până la primul caracter *linie nouă* (\n), în funcţie de ce întâlneşte mai întâi.

Multe programe vor utiliza funcţia *sizeof* pentru a specifica dimensiunea bufferului, de exemplu, *sizeof(sir)*. În sfârşit, parametrul *flux* precizează fişierul din care funcţia *fgets* trebuie să citească şirul de caractere. Anterior, trebuie să deschideţi fluxul utilizând *fopen* sau un indicator predefinit, cum ar fi *stdin*. Dacă funcţia *fgets* reuşeşte să citească informaţiile din fişier, ea va returna un pointer la şirul de caractere. Dacă apare o eroare sau dacă funcţia a ajuns la sfârşitul fişierului, funcţia *fgets* va returna *NULL*.

SCRIEREA LINIILOR DE TEXT

C/C++ 443

În secţiunea 442 aţi învăţat că de obicei programele dumneavoastră citesc din fişier, linie cu linie. Atunci când se scrie într-un fişier, programele dumneavoastră vor scrie, de regulă, linie cu linie. Pentru a scrie un şir de caractere într-un fişier, programele dumneavoastră pot utiliza funcţia *fputs*, ca mai jos:

```
#include <stdio.h>

int fputs(const char *sir, FILE *flux);
```

Funcţia *fputs* scrie caracterele în şirul specificat până la terminatorul de şir, *NULL* ('\0'). Dacă funcţia *fputs* reuşeşte să scrie şirul, ea va returna o valoare pozitivă. Dacă apare o eroare, funcţia *fputs* va returna constanta *EOF*.

444 UTILIZAREA FUNCȚIILOR FGETS ȘI FPUTS



În secțiunile 442 și 443 ați învățat că programele dumneavoastră pot utiliza funcțiile *fgets* și *fputs* pentru a citi și scrie date din/în fișiere. Următorul program, *textcop.c*, utilizează funcțiile *fgets* și *fputs* pentru a copia conținutul primului fișier specificat în linia de comandă în cel de al doilea fișier specificat în linia de comandă:

```
#include <stdio.h>

void main(int argc, char **argv)
  {
    FILE *intrare, *iesire;
    char sir[256];

    if ((intrare = fopen(argv[1], "r")) == NULL)
      printf("Eroare la deschiderea %s\n", argv[1]);
    else if ((iesire = fopen(argv[2], "w")) == NULL)
      {
        printf("Eroare la deschiderea %s\n", argv[2]);
        fclose(intrare);
      }
    else
      {
        while (fgets(sir, sizeof(sir), intrare))
          fputs(sir, iesire);
        fclose(intrare);
        fclose(iesire);
      }
  }
```

Așa cum vedeți, programul deschide un fișier de intrare și unul de ieșire, apoi citește și scrie textul până când funcția *fgets* întâlnește sfârșitul fișierului (funcția *fgets* returnează *NULL*). De exemplu, pentru a copia conținutul fișierului *test.dat* în *test.sav*, puteți folosi programul *textcop.c* în felul următor:

```
C:\> TEXTCOP TEST.DAT TEST.SAV <ENTER>
```

445 FORȚAREA CONVERSIEI FIȘIERULUI BINAR



Așa cum ați învățat, multe compilatoare de C utilizează variabila globală *_fmode* pentru a determina modul de acces text sau binar la fișier. În modul text, funcțiile C de bibliotecă run-time convertesc caracterele *avans rând* în combinații *retur de car* și *avans rând* și viceversa. Așa cum ați învățat, prin stabilirea variabilei *_fmode* la *O_TEXT* sau *O_BINARY* puteți controla modul de acces. În plus, prin plasarea lui *t* sau *b* în modul de acces specificat în *fopen*, puteți stabili modul de acces pentru modul text sau modul binar. De exemplu, următorul apel al funcției *fopen* deschide fișierul *numefis.ext* pentru acces cu citire în mod binar:

```
if ((pointer_fisier = fopen("NUMEFIS.EXT", "rb")) == NULL)
```

SĂ ÎNȚELEGEM DE CE TEXTCOP NU POATE COPIA FIȘIERE BINARE

C/C++ 446

În secțiunea 444 a fost prezentat programul *textcop.c* care copiază conținutul primului fișier specificat în linia de comandă în cel de al doilea fișier. Dacă încercați să utilizați *textcop* pentru a copia un fișier binar, cum ar fi un fișier *.exe*, operația de copiere nu va reuși. Când funcția *fgets* citește un fișier text, ea consideră caracterul CTRL+Z (caracterul ASCII 26) ca fiind sfârșitul fișierului. Deoarece un fișier binar e posibil să conțină una sau mai multe apariții ale caracterului ASCII 26, funcția *fgets* va termina operația de copiere la prima lui apariție. Dacă vreți să copiați un fișier executabil sau un alt fișier binar, trebuie să utilizați rutinele limbajului C de I/O de nivel inferior.

TESTAREA SFÂRȘITULUI DE FIȘIER

C/C++ 447

Așa cum ați învățat, când funcția *fgets* întâlnește sfârșitul de fișier, ea returnează *NULL*. De asemenea, când funcția *fgetc* ajunge la sfârșitul fișierului, ea returnează *EOF*. Uneori programele dumneavoastră trebuie să determine dacă pointerul de fișier este la sfârșitul fișierului, înainte de executarea unei anumite operații. În aceste situații, programele dumneavoastră pot apela funcția *feof*, prezentată mai jos:

```
#include <stdio.h>

int feof(FILE *flux);
```

Dacă pointerul de fișier specificat este la sfârșitul fișierului, funcția *feof* va returna o valoare diferită de 0 (adevărat). Dacă nu a ajuns încă la sfârșitul fișierului, funcția *feof* va returna 0 (fals). Următoarea buclă citește și afișează caracterele din fișierul care corespunde pointerului de fișier *intrare*:

```
while (! feof(intrare))
   fputc(fgetc(intrare), stdout);
```

***Observație:** După ce o funcție, cum ar fi **fgetc**, stabilește indicatorul de sfârșit de fișier, el rămâne așa până când programul închide fișierul sau apelează funcția **rewind**.*

UTILIZAREA FUNCȚIEI UNGETC

C/C++ 448

Multe programe, un compilator de exemplu, citesc caractere dintr-un fișier, unul câte unul, până când întâlnesc un caracter specific (un delimitator sau un simbol). După ce programul găsește caracterul, programul execută anumite prelucrări. După încheierea acestor prelucrări, programul continuă să citească din fișier. În funcție de structura fișierului pe care îl citește programul, puteți anula citirea unui caracter. În aceste situații, programul poate utiliza funcția *ungetc* al cărei format e prezentat mai jos:

```
#include <stdio.h>

int ungetc(int caracter, FILE *flux);
```

Funcția *ungetc* plasează caracterul specificat înapoi în bufferul fișierului. Funcția nu are efect decât asupra unui caracter. Dacă apelați funcția *ungetc* de două ori succesiv, al doilea

caracter îl va rescrie pe primul anulat. Funcția *ungetc* plasează caracterul specificat în membrul *hold* al structurii *FILE*. În schimb, următoarea operație de citire a fișierului, va include caracterul respectiv.

449 CITIREA DATELOR FORMATATE DIN FIȘIER



Ați învățat cum să utilizați funcția *fprintf* pentru a scrie ieșiri formatate într-un fișier. Într-un mod similar, funcția *fscanf* vă permite să citiți date formatate din fișier, așa cum funcția *scanf*, despre care ați învățat anterior, vă permite să citiți date formatate de la tastatură. Implementarea funcției *fscanf* este următoarea:

```
#include <stdio.h>

int fscanf(FILE *flux, const char *format[, adresa_variabila,
   ...]);
```

Parametrul *flux* este un pointer la fișierul din care doriți să citiți cu funcția *fscanf*. Parametrul *format* precizează formatul datelor, utilizând același caracter de control ca funcția *scanf*. În sfârșit, parametrul *adresa_variabila* specifică adresa la care doriți să citiți datele. Punctele de suspensie (...) care urmează parametrului *adresa_variabila* indică faptul că puteți utiliza mai multe adrese separate prin virgulă.

La încheierea execuției, funcția *fscanf* returnează numărul de câmpuri citite. Dacă funcția *fscanf* întâlnește sfârșitul de fișier, funcția returnează constanta *EOF*. Următorul program, *fscanf.c*, deschide fișierul *date.dat* pentru ieșire, scrie ieșirea formatată în fișier, utilizând funcția *fprintf*, închide fișierul și apoi, îl redeschide pentru intrare, citindu-i conținutul cu funcția *fscanf*:

```
#include <stdio.h>

void main(void)
  {
    FILE *pointer_fisier;

    int varsta;
    float salariu;
    char nume[64];

    if ((pointer_fisier = fopen("DATE.DAT", "w")) == NULL)
      printf("Eroare la deschiderea DATE.DAT pentru iesire\n");
    else
      {
        fprintf(pointer_fisier, "33 35000.0 Kris");
        fclose(pointer_fisier);

        if ((pointer_fisier = fopen("DATE.DAT", "r")) == NULL)
          printf("Eroare la deschiderea DATE.DAT pentru intrare\n");
        else
          {
            fscanf(pointer_fisier, "%d %f %s", &varsta, &salariu,
              nume);
```

```
            printf("Varsta %d Salariu %f Nume %s\n", varsta,
              salariu, nume);
            fclose(pointer_fisier);
          }
      }
  }
```

POZIȚIONAREA POINTERULUI DE FIȘIER PE BAZA LOCAȚIEI SALE CURENTE

C/C++ 450

Ați învățat că un pointer de fișier conține un pointer de poziție care urmărește poziția curentă în cadrul fișierului. Când știți formatul fișierului, puteți să deplasați pointerul de poziție la o anumită locație înainte de a începe citirea fișierului. De exemplu, primii 256 de octeți ai fișierului dumneavoastră pot conține informații de antet, pe care nu doriți să le citiți. În aceste cazuri, programul dumneavoastră poate folosi funcția *fseek* pentru a poziționa pointerul de fișier, în formatul de mai jos:

```
#include <stdio.h>

int fseek(FILE *flux, long deplasament, int relativ_la);
```

Parametrul *flux* specifică pointerul de fișier pe care vreți să îl poziționați. Parametrii *deplasament* și *relativ_la* se combină pentru a preciza poziția dorită. Parametrul *deplasament* conține octetul de deplasament din fișier. Parametrul *relativ_la* precizează locația din fișier de la care funcția *fseek* trebuie să aplice deplasamentul. Tabelul 450 prezintă valorile pe care le puteți folosi pentru parametrul *relativ_la*.

Constantă	Semnificație
SEEK_CUR	De la poziția curentă a fișierului
SEEK_SET	De la începutul fișierului
SEEK_END	De la sfârșitul fișierului

Tabelul 450 *Poziția în fișier de la care funcția **fseek** poate aplica un deplasament.*

Pentru a poziționa pointerul de fișier imediat după primii 256 de octeți cu informații de antet din fișier, puteți utiliza funcția *fseek* în felul următor:

```
fseek(pointer_fisier, 256, SEEK_SET); // Deplasamentul 0 este
                                       // la inceputul fisierului
```

Dacă reușește, funcția *fseek* returnează valoarea 0. Dacă apare o eroare, funcția returnează o valoare diferită de 0.

OBȚINEREA INFORMAȚIILOR DESPRE INDICATORUL DE FIȘIER

C/C++ 451

Atunci când operați cu un indicator de fișier, puteți să cunoașteți amănunte despre fișierul corespunzător, cum ar fi, de exemplu, unitatea de disc pe care este stocat fișierul. În acest caz, programele dumneavoastră pot utiliza funcția *fstat*, care are următorul format:

```
#include <sys\stat.h>
int fstat(int indicator, struct stat *buffer);
```

Funcția atribuie date specifice despre fișier unei structuri de tip *stat* definită în fișierul *stat.h*, ca mai jos:

```
struct_stat
  {
    short st_dev;   // numarul unitatii de disc
    short st_ino;   // neutilizat de DOS
    short st_mode;  // mod deschidere fisier
    short st_nlink; // intotdeauna 1
    short st_uid;   // identificator de utilizator -- neutilizat
    short st_gid;   // identificator de utilizator -- neutilizat
    short st_rdev;  // la fel ca st_dev
    long st_size;   // dimensiune fisier in octeti
    long st_atime;  // ora ultimei deschideri a fisierului
    long st_mtime;  // la fel ca st_atime
    long st_ctime;  // la fel ca st_atime
  };
```

Dacă funcția *fstat* reușește, ea returnează valoarea 0. Dacă apare o eroare, funcția returnează valoarea -1 și stabilește variabila globală *errno* la EBADF (indicator de fișier greșit). Următorul program, *autoinfo.c*, utilizează funcția *fstat* pentru a afișa data și ora ultimei modificări a fișierului *autoexec.bat*, precum și dimensiunea sa:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <time.h>

void main(void)
  {
    int indicator;
    struct stat buffer;

    if ((indicator = open("\\AUTOEXEC.BAT", O_RDONLY)) == -1)
     printf("Eroare la deschiderea \\AUTOEXEC.BAT\n");
    else
      {
        if (fstat(indicator, &buffer))
         printf("Eroare la obtinerea informatiilor din fisier\n");
        else
         printf("AUTOEXEC.BAT are %ld octeti Ultima utilizare
            %s\n",
           buffer.st_size, ctime(&buffer.st_atime));
        close(indicator);
      }
  }
```

Observaţie: *Secţiunea 1465 prezintă în detaliu modul în care puteţi obţine marcarea orei şi datei într-un fişier Windows.*

REDESCHIDEREA UNUI FLUX DE FIŞIER

C/C++ 452

Deoarece programele dumneavoastră operează cu fişiere, puteţi să suprascrieţi un pointer de fişier deschis. De exemplu, DOS nu pune la dispoziţie un mod de a redirecta ieşirea unui indicator de fişier *stderr* de la linia de comandă. Puteţi însă, rescrie în cadrul programului dumneavoastră, destinaţia pointerului de fişier *stderr* prin redeschiderea lui, apelând funcţia *freopen*:

```
#include <stdio.h>
FILE *freopen(const char *numefisier, const char *mod_acces,
  FILE *flux);
```

Funcţia *freopen* este similară cu funcţia *fopen*, cu excepţia faptului că pasaţi funcţiei un pointer de fişier a cărui valoare doriţi să o suprascrieţi. Dacă funcţia reuşeşte, ea returnează un pointer la fluxul originar de fişier. Dacă apare o eroare, funcţia *freopen* returnează *NULL*. Următorul program, *nustderr.c*, de exemplu, redirectează funcţiile *stderr* către fişierul *standard.err*, şi nu spre ecran:

```
#include <stdio.h>
void main(void)
 {
   if (freopen("STANDARD.ERR", "w", stderr))
     fputs("stderr a fost redirectat", stderr);
   else
     printf("Eroare la redeschidere\n");
 }
```

MATRICELE

C/C++ 453

Aşa cum aţi învăţat, un tip descrie setul de valori pe care o variabilă le poate avea şi un set de operaţii pe care programele dumneavoastră le pot executa cu variabila respectivă. Cu excepţia şirurilor de caractere, toate tipurile pe care le-aţi studiat până acum pot avea numai o valoare. Pe măsură ce programele dumneavoastră încep să execute operaţii mai utile, este posibil ca o variabilă să conţină mai multe valori. De exemplu, variabila *punctaje* poate să reţină punctajul la un test pentru 100 de elevi. De asemenea, variabila *salarii* poate reţine salariul fiecărui angajat al unei firme. O matrice (tablou) este o structură de date care poate să păstreze mai multe valori de acelaşi tip. De exemplu, puteţi să creaţi o matrice care să reţină 100 de valori de tip *int* şi o a doua matrice care să reţină 25 de valori de tip *float*.

Fiecare valoare pe care o atribuiţi unei matrice trebuie să fie de acelaşi tip cu tipul matricei. În această secţiune veţi învăţa cum să creaţi şi să lucraţi cu matricele în programele dumneavoastră. După ce lucraţi cu una sau două matrice, veţi putea observa că ele sunt uşor de înţeles. Dacă deja v-aţi acomodat cu şirurile de caractere, curând vă veţi simţi la fel şi lucrând cu matricele. Amintiţi-vă, un şir de caractere nu este decât o matrice de caractere.

454 DECLARAREA UNEI MATRICE

C/C++

În secțiunea 453 ați învățat că o matrice este o variabilă care poate să păstreze mai multe valori de același tip. Pentru a declara o matrice, trebuie să specificați tipul dorit (cum ar fi *int*, *float* sau *double*), și de asemenea, dimensiunea matricei. Pentru a preciza dimensiunea unei matrice, trebuie să plasați între paranteze drepte numărul de valori pe care matricea respectivă poate să le păstreze, care urmează imediat după numele matricei. Următoarea declarare, de exemplu, creează o matrice cu numele *punctaje* care poate păstra 100 de punctaje la test, de tip *int*:

```
int punctaje[100];
```

Într-un mod asemănător, următoarea declarare creează o matrice de tip *float* care conține 50 de salarii:

```
float salarii[50];
```

Atunci când declarați o matrice, compilatorul de C alocă suficientă memorie pentru a putea reține toate elementele. Prima intrare este la locația 0. De exemplu, în matricele *punctaje* și *salarii*, următoarea instrucțiune atribuie valorile 80 și, respectiv, 35000 primului element al matricelor:

```
punctaje[0] = 80;
salarii[0] = 35000.0;
```

Deoarece primul element al matricei începe la deplasament 0, ultimul element al matricei va apărea la o locație cu 1 mai mică decât dimensiunea matricei. Date fiind matricele anterioare, *punctaje* și *salarii*, următoarea instrucțiune atribuie valorile ultimului element al fiecărei matrice:

```
punctaje[99] = 75;
salarii[49] = 24000.0;
```

455 VIZUALIZAREA UNEI MATRICE

C/C++

Așa cum ați învățat, o matrice este o variabilă care poate să păstreze mai multe valori de același tip. Pentru a vă ajuta să înțelegeți mai bine cum păstrează informația o matrice, să studiem următoarea declarare a unei matrice:

```
char sir[64];
float salarii[50];
int punctaje[100];
long planete[13];
```

După ce atribuiți valori fiecărei matrice, ele vor rămâne în memorie într-o manieră similară cu cea prezentată în figura 455.

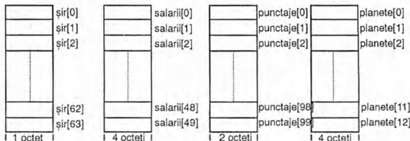


Figura 455 *Păstrarea valorilor în matrice.*

După cum puteţi vedea, fiecare prima valoare a unei matrice se păstrează la deplasament 0. În primul capitol al acestei cărţi, aţi învăţat că o variabilă este un nume pe care îl atribuiţi uneia sau mai multor locaţii de memorie. Într-o matrice, puteţi avea un număr mare de locaţii de memorie care corespund unei singure matrice.

CERINŢELE DE STOCARE ALE UNEI MATRICE

C/C++ 456

Aşa cum aţi învăţat, o matrice este denumirea dată unei colecţii de valori de acelaşi tip. Atunci când declaraţi o matrice, compilatorul de C alocă suficientă memorie pentru a putea păstra numărul de valori pe care îl precizaţi. Dimensiunea reală a memoriei pe care compilatorul o alocă depinde de tipul matricei. De exemplu, o matrice de 100 de elemente de tip *int* va cere, de obicei, 100*2 sau 200 de octeţi de memorie. O matrice de 100 de elemente de tip *float*, însă, va cere 100*4 sau 400 de octeţi. Următorul program, *dimtabl.c*, utilizează operatorul *sizeof* al limbajului C pentru a afişa volumul de memorie solicitat de diferitele tipuri de matrice:

```
#include <stdio.h>

void main(void)
  {
    int punctaje[100];
    float salarii[100];
    char sir[100];
    printf("Octeti pentru pastrarea punctaje[100] de tip int
      %d octeti\n", sizeof(punctaje));
    printf("Octeti pentru pastrarea salarii[100] de tip float
      %d octeti\n", sizeof(salarii));
    printf("Octeti pentru pastrarea sir[100] de tip char
      %d octeti\n", sizeof(sir));
  }
```

Atunci când compilaţi şi executaţi programul *dim tabl.c*, ecranul dumneavoastră va afişa următorul rezultat:

```
Octeti pentru pastrarea punctaje[100] de tip int 200 octeti
Octeti pentru pastrarea salarii[100] de tip float 400 octeti
Octeti pentru pastrarea sir[100] de tip char 100 octeti
C:\>
```

457 INIȚIALIZAREA UNEI MATRICE



De-a lungul acestei cărți, multe programe au inițializat un șir de caractere astfel:

```
char titlu[] = "Totul despre C/C++";
char sectiune[64] = "Tablouri";
```

În primul caz, compilatorul de C va aloca 24 de octeți pentru a stoca șirul. În al doilea caz, compilatorul va aloca o matrice de 64 de octeți, inițializând primele șapte caractere literelor cuvântului "tablouri" și caracterului *NULL*. Majoritatea compilatoarelor vor inițializa, de asemenea, și restul locațiilor la *NULL*. Atunci când declarați o matrice de alte tipuri, puteți să inițializați matricea în același mod. De exemplu, următoarea instrucțiune inițializează matricea de întregi *punctaje* cu valorile 80, 70, 90, 85 și 80:

```
int punctaje[5] = {80, 70, 90, 85, 80};
```

Când veți atribui valorile inițiale matricei, trebuie să transmiteți valorile între acolade ({}). În cazul anterior, dimensiunea matricei este egală cu numărul valorilor atribuite matricei. Următoarea instrucțiune, însă, atribuie patru valori *în virgulă mobilă* unei matrice care poate păstra 64 de valori:

```
float salarii[64] = {25000.0, 32000.0, 44000.0, 23000.0};
```

În funcție de compilatorul dumneavoastră, puteți atribui valoarea 0 elementelor pentru care programul dumneavoastră nu atribuie în mod explicit o valoare. Ca regulă, însă, nu trebuie să presupuneți că va inițializa compilatorul celelalte elemente. Mai mult, dacă nu precizați dimensiunea matricei, compilatorul va aloca atâta memorie cât este necesară pentru a păstra numai valorile pe care le-ați specificat. Următoarea declarare a unei matrice, de exemplu, creează o matrice cu dimensiunea suficient de mare pentru a putea păstra trei valori de tip *long*:

```
long planete[] = {1234567L, 654321L, 1221311L};
```

458 ACCESAREA ELEMENTELOR UNEI MATRICE



Valorile stocate într-o matrice sunt denumite *elementele matricei*. Pentru a accesa un element al unei matrice, precizați numele matricei și elementul pe care îl doriți. Următorul program, *elemente.c*, inițializează matricea *punctaje* și apoi utilizează funcția *printf* pentru a afișa valorile elementelor:

```
#include <stdio.h>

void main(void)
  {
    int punctaje[5] = {80, 70, 90, 85, 80};
    printf("Valorile tabloului\n");
    printf("punctaje[0] %d\n", punctaje[0]);
    printf("punctaje[1] %d\n", punctaje[1]);
    printf("punctaje[2] %d\n", punctaje[2]);
    printf("punctaje[3] %d\n", punctaje[3]);
```

```
    printf("punctaje[4] %d\n", punctaje[4]);
  }
```

Atunci când compilaţi şi executaţi programul *elemente.c*, ecranul dumneavoastră va afişa următorul rezultat:

```
Valorile tabloului
punctaje[0] = 80
punctaje[1] = 70
punctaje[2] = 90
punctaje[3] = 85
punctaje[4] = 80
C:\>
```

După cum puteţi vedea, pentru a accesa un anumit element al unei matrice, specificaţi numărul elementului pe care îl doriţi între paranteze drepte, după numele matricei.

CICLAREA PRIN ELEMENTELE UNEI MATRICE

C/C++ 459

În secţiunea 458 aţi utilizat valorile de la 0 la 4 pentru a fişa elementele matricei *punctaje*. Atunci când faceţi referire la mai multe elemente ale matricei, specificând numărul fiecăruia dintre elemente în mod individual, poate lua mult timp. Ca alternativă, programele dumneavoastră pot utiliza o variabilă pentru referinţa la elementele matricei. De exemplu, presupunând că variabila *i* conţine valoarea 2, următoarea instrucţiune va atribui matricei pentru *tablou[2]* valoarea 80:

```
i = 2;

tablou[i] = 80;
```

Următorul program, *unmatrice.c*, utilizează variabila *i* şi bucla *for* pentru a afişa elementele matricei *punctaje*:

```
#include <stdio.h>

void main(void)
  {
    int punctaje[5] = {80, 70, 90, 85, 80};
    int i;

    printf("Valorile tabloului\n");

    for (i = 0; i < 5; i++)
      printf("punctaje[%d] %d\n", i, punctaje[i]);
  }
```

UTILIZAREA CONSTANTELOR PENTRU DEFINIREA UNEI MATRICE

C/C++ 460

Aşa cum aţi învăţat, când programele dumneavoastră lucrează cu matrice, trebuie să specificaţi dimensiunea ei. De exemplu, următorul program, *5_val.c*, declară o matrice de cinci valori şi apoi utilizează bucla *for* pentru a afişa valorile matricei:

```
#include <stdio.h>

void main(void)
  {
    int val[5] = {80, 70, 90, 85, 80};
    int i;
    for (i = 0; i < 5; i++)
      printf("valorile[%d] %d\n", i, val[i]);
  }
```

Presupunem, de exemplu, că mai târziu veți dori să modificați programul *5_val.c* astfel încât să accepte 10 valori. Pentru aceasta trebuie nu numai să modificați declararea matricei, ci și bucla *for*. Cu cât trebuie să aduceți programului mai multe modificări, cu atât va fi mai mare șansa de a produce erori. Ca alternativă, programele dumneavoastră pot să declare matricele utilizând constante. Următorul program, *5_const.c*, declară o matrice utilizând constanta *ARRAY_SIZE*. După cum vedeți, programul nu numai că utilizează constanta la declararea matricei, dar o utilizează, de asemenea, pentru condiția de sfârșit a buclei *for*:

```
#include <stdio.h>

#define ARRAY_SIZE 5

void main(void)
  {
    int val[ARRAY_SIZE] = {80, 70, 90, 85, 80};
    int i;

    for (i = 0; i < ARRAY_SIZE; i++)
      printf("valorile[%d] %d\n", i, val[i]);
  }
```

Dacă trebuie să modificați mai târziu dimensiunile matricei, puteți să modificați valoarea atribuită constantei *ARRAY_SIZE* astfel ca programul să actualizeze automat buclele care controlează matricea și dimensiunea ei.

461 TRANSMITEREA UNEI MATRICE UNEI FUNCȚII



Așa cum ați învățat, o matrice este o variabilă care poate păstra mai multe valori de același tip. Ca orice variabilă, programele dumneavoastră pot transmite o matrice unei funcții. Atunci când declarați o funcție care are ca parametru o matrice, trebuie să informați compilatorul despre acest lucru. De exemplu, următorul program, *func_tab.c*, utilizează funcția *o_matrice* pentru a afișa valorile unei matrice. După cum puteți observa, programul transmite funcției atât matricea, cât și numărul de elemente conținute de matrice, cum arătăm în continuare:

```
#include <stdio.h>

void un_tablou(int val[], int nr_de_elemente)
  {
    int i;
```

```
    for (i = 0; i < nr_de_elemente; i++)
      printf("%d\n", val[i]);
  }
void main(void)
  {
    int punctaje[5] = {70, 80, 90, 100, 90};
    un_tablou(punctaje, 5);
  }
```

Atunci când funcţia primeşte o matrice ca parametru, programul dumneavoastră nu trebuie să specifice dimensiunea ei în declararea parametrului. În cazul funcţiei *o_matrice*, parantezele drepte care urmează numele variabilei *val* informează compilatorul că parametrul este o matrice. Dacă nu ştie că parametrul este o matrice, compilatorul nu se îngrijeşte de dimensiunea matricei pe care programul dumneavoastră o transmite funcţiei.

DIN NOU DESPRE TRANSMITEREA UNEI MATRICE CĂTRE O FUNCŢIE

C/C++ 462

În secţiunea 461 aţi învăţat că atunci când declaraţi un parametru formal ca fiind o matrice, nu trebuie să declaraţi dimensiunea matricei. În schimb, puteţi numai să plasaţi parantezele drepte. Următorul program, *paramtab.c*, transmite trei matrice diferite (de diferite dimensiuni) funcţiei *o_matrice*:

```
#include <stdio.h>
void un_tablou(int val[], int nr_de_elemente)
  {
    int i;
    printf("Afiseaza %d valori\n", nr_de_elemente);
    for (i = 0; i < nr_de_elemente; i++)
      printf("%d\n", val[i]);
  }
void main(void)
  {
    int punctaje[5] = {70, 80, 90, 100, 90};
    int contor[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int mic[2] = {-33, -44};
    un_tablou(punctaje, 5);
    un_tablou(contor, 10);
    un_tablou(mic, 2);
  }
```

Atunci când compilaţi şi executaţi programul *paramtab.c*, ecranul dumneavoastră va afişa fiecare dintre valorile matricei. Aşa cum aţi învăţat, funcţia nu are nevoie de dimensiunea matricei. Însă, observaţi că matricele pe care programul *paramtab.c* le transmite funcţiei sunt toate de tip *int*. Dacă încercaţi să transmiteţi unei funcţii o matrice de tip *float*, compilatorul va genera o eroare.

463 DIFERENȚA ÎNTRE MATRICELE ȘIRURI ȘI MATRICE

C/C++

Multe dintre secțiunile prezentate de-a lungul acestei cărți au transmis șiruri de caractere unor funcții. În majoritatea cazurilor, funcțiile nu specificau dimensiunea șirului. De exemplu, următoarea instrucțiune utilizează funcția *strupr* pentru a converti un șir de caractere în majuscule:

```
char titlu[64] = "Totul despre C/C++";
strupr(titlu);
```

Așa cum ați învățat, în limbajul C, caracterul *NULL* reprezintă sfârșitul unui șir de caractere. De aceea, funcțiile pot căuta caracterul *NULL* în elementele matricei, pentru a determina unde se sfârșește matricea. Matricele de alte tipuri, cum ar fi *int*, *float* sau *long*, nu au un caracter de sfârșit echivalent. De aceea, trebuie, de obicei, să transmiteți funcției care lucrează cu matrice numărul de elemente pe care le conține matricea.

464 TRANSMITEREA MATRICELOR ÎN STIVĂ

C/C++

Câteva dintre secțiunile anterioare au abordat problema transmiterii unei matrice ca parametru al unei funcții. Atunci când transmiteți o matrice unei funcții, compilatorul de C plasează numai adresa primului element al matricei în stivă. Figura 464, de exemplu, ilustrează matricea *punctaje* și apelul funcției *arata_tablou* care utilizează parametrul *punctaje*:

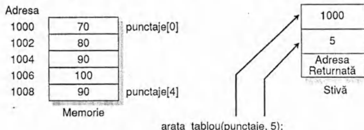


Figura 464 *Când transmiteți o matrice ca parametru, compilatorul de C plasează adresa de început a matricei în stivă.*

După cum puteți vedea, compilatorul de C plasează numai adresa de început a matricei în stivă. De asemenea, rețineți că funcția nu primește de la compilator informații în legătură cu dimensiunea matricei.

465 DETERMINAREA NUMĂRULUI DE ELEMENTE CARE POT FI PĂSTRATE DE O MATRICE

C/C++

Așa cum ați învățat, volumul real de memorie pe care o matrice îl poate consuma va diferi, în funcție de tipul matricei. Dacă lucrați în mediu DOS, volumul de memorie pe care matricele dumneavoastră îl pot consuma va depinde de modelul curent de memorie. În general, o matrice nu poate consuma mai mult de 64 Kb de spațiu. Următorul program, *preamare.c*, de

exemplu, este posibil să eșueze la compilare pentru că matricea consumă prea mult memorie:

```
void main(void)
  {
    char sir[66000L];        // 66000 octeti
    int val[33000L];         // 33000 * 2 = 66000 octeti
    float numere[17000];     // 17000 * 4 = 68000 octeti
  }
```

Observație: *Deoarece Windows utilizează modelul de memorie virtuală pentru a gestion. memoria, el nu pune limite dimensiunii matricei ca în cazul programelor C care rulează sub DOS. De exemplu, în Windows puteți să declarați șiruri de caractere (matrice de caractere) de mărimea constantei INT_MAX (2 147 483 647). Dacă, însă, încercați să declarați o variabilă de dimensiune superioară limitei, într-o fereastră DOS, veți produce eroare de stivă (**stack fault**) și Windows va închide fereastra.*

UTILIZAREA MODELULUI DE MEMORIE HUGE PENTRU MATRICE MARI

C/C++ 466

Dacă o matrice consumă un volum de memorie care excede 64 Kb, puteți să indicați multor dintre compilatoarele care funcționează sub DOS să utilizeze *modelul de memorie hug* tratând matricea ca pointer și incluzând cuvântul cheie *huge* în cadrul declarării, cum arătăm în continuare:

```
float huge val[17000];
```

Următorul program, *buge.c*, creează o matrice de tip *float*, de mari dimensiuni:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
  {
    int i;
    float huge *val;

    if ((val = (float huge *) halloc (17000, sizeof(float)))
      == NULL)
     printf ("Eroare de alocare a matricei huge\n");
    else
      {
       printf("Completeaza matricea\n");
       for (i = 0; i < 17000; i++)
         val[i] = i * 1.0;
       for (i = 0; i < 17000; i++)
         printf ("%8.1f ", val[i]);
       hfree(val);
      }
  }
```

Observaţie: *Deoarece Windows utilizează modelul de memorie virtuală pentru a gestiona memoria, el nu limitează dimensiunile matricei, cum se întâmplă cu programele de C sub DOS. De exemplu, puteţi declara o matrice de tipul* ***unsigned char*** *în Windows, de mărimea constantei INT_MAX (2 147 483 647), fără a folosi cuvântul cheie* ***huge.*** *Dacă, însă, încercaţi să declaraţi o variabilă cu dimensiunea superioară limitei în cadrul unei ferestre DOS, fără a folosi cuvântul cheie* ***huge****, veţi produce o eroare de stivă (****stack fault****) şi Windows va închide fereastra.*

467 ALEGEREA ÎNTRE MATRICE ŞI MEMORIA DINAMICĂ



Pe măsură ce vă acomodaţi cu limbajul C şi cu modalităţile de utilizare a pointerilor în cadrul acestuia, puteţi să începeţi să utilizaţi mai puţin matricele, în schimb veţi aloca memorie dinamică atunci când veţi avea nevoie. Există câteva avantaje şi dezavantaje de care trebuie să ţineţi seama pentru a decide dacă să folosiţi memoria dinamică sau o matrice. În primul rând, mulţi programatori găsesc matricele mai simplu de înţeles şi utilizat. Prin urmare, programul dumneavoastră poate el însuşi să fie mai uşor de urmărit de un alt programator. În al doilea rând, deoarece compilatorul alocă spaţiu pentru matrice, programele dumneavoastră vor evita suprasarcina asociată cu alocarea memoriei dinamice. Ca urmare, un program care se bazează pe utilizarea matricelor poate să se execute ceva mai repede.

Aşa cum aţi învăţat, însă, atunci când declaraţi o matrice, trebuie să specificaţi dimensiunea sa. Dacă nu ştiţi de ce dimensiune aveţi nevoie, puteţi să alocaţi o matrice mai mare decât ar fi necesar. Ca urmare, puteţi să faceţi risipă de spaţiu de memorie. Pe de altă parte, dacă dimensiunea matricei este prea mică, trebuie să editaţi programul, să modificaţi dimensiunea matricei şi să recompilaţi programul.

Atunci când declaraţi o matrice în cadrul programelor dumneavoastră, reţineţi că puteţi executa aceeaşi procedură şi prin alocarea memoriei dinamice. Aşa cum veţi învăţa în capitolul despre pointeri al acestei cărţi, puteţi adresa alocarea dinamică a memoriei utilizând indecşii de matrice şi să eliminaţi astfel confuzia datorată pointerilor care dezorientează frecvent pe noii programatori în C. Pentru că majoritatea sistemelor de operare permit programelor să aloce memorie dinamică foarte rapid, s-ar putea să preferaţi flexibilitatea şi oportunităţile superioare de gestionare a memoriei pe care alocarea dinamică a memoriei le oferă faţă de matrice, în ciuda uşoarei supraîncărcări a sistemului pe care o produce.

468 MATRICELE MULTIDIMENSIONALE



Aşa cum aţi învăţat, o matrice este o variabilă care poate păstra mai multe valori de acelaşi tip. În toate exemplele prezentate până acum, matricele au constat într-un singur rând de date. Totuşi, limbajul C acceptă matrice bi-, tri- şi multidimensionale. Cea mai bună cale de a vizualiza o matrice bidimensională este un tabel cu rânduri şi coloane. Dacă o matrice conţine trei dimensiuni, consideraţi-o ca pe mai multe pagini, fiecare din ele conţinând tabele bidimensionale, cum arată figura 468.

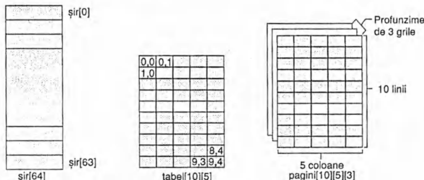


Figura 468 *Modelele logice de matrice multidimensionale.*

Următoarele declarări ale unor matrice creează matricele arătate în figura 468:

```
char sir[64];
int tabel[10][5];
float pagini[10][5][3];
```

RÂNDURILE ŞI COLOANELE

C/C++469

Aşa cum aţi învăţat, limbajul C acceptă matrice multidimensionale care sunt similare unui tabel de valori. Atunci când lucraţi cu o matrice bidimensională, gândiţi matricea ca pe un tabel cu rânduri şi coloane. Rândurile tabelului merg de la stânga spre dreapta, în timp ce coloanele merg de sus în jos pe pagină, cum se arată în figura 469.

Linia 0
Linia 1
Linia 2

Coloana 0 | Coloana 1 | Coloana 2

Figura 469 *Rândurile şi coloanele unei matrice bidimensionale.*

Atunci când declaraţi o matrice bidimensională, prima valoare specificată prezintă numărul de rânduri şi a doua valoare, numărul de coloane:

```
int tabel[2][3];
```

470 ACCESAREA ELEMENTELOR UNEI MATRICE BIDIMENSIONALE



Așa cum ați învățat, cel mai bine puteți să vizualizați o matrice bidimensională ca pe un tabel cu rânduri și coloane. Pentru a adresa un element anume al matricei, trebuie să precizați rândul și coloana corespunzătoare poziției acestuia. Figura 470 ilustrează instrucțiunile care accesează elementele specificate din matricea *tabel*.

tabel [0][0] = 0;
tabel [0][2] = 2;
tabel [1][1] = 5;
tabel [2][2] = 14;
tabel [3][2] = 77;

0		2
	5	
		14
		77

tabel [4][3];

Figura 470 *Pentru a accesa elementele dintr-o matrice bidimensională, trebuie să precizați poziția pe rânduri și coloane a elementului.*

După cum puteți vedea, atunci când accesați o matrice bidimensională, deplasamentul rândurilor și coloanelor începe la 0.

471 INIȚIALIZAREA ELEMENTELOR ÎNTR-O MATRICE BIDIMENSIONALĂ



În secțiunea 457 ați învățat că, pentru a inițializa elementele unei matrice, puteți plasa valorile elementelor între acolade, după declararea matricei. Următoarea instrucțiune utilizează aceeași tehnică pentru a inițializa o matrice bidimensională. Însă, în acest caz, instrucțiunea specifică valorile pentru fiecare rând al matricei în parte, între acolade:

```
int tabel[2][3] = {{1, 2, 3},
                   {4, 5, 6}};
```

Compilatorul va inițializa elementele matricei ca în figura 471.

1	2	3
4	5	6

tabel [2][3];

Figura 471 *Inițializarea elementelor unei matrice bidimensionale.*

Într-o manieră similară, următoarea instrucțiune inițializează elementele unei matrice mai mare:

```
int vanzari[4][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15},
                    {16, 17, 18, 19, 20}};
```

DETERMINAREA CONSUMULUI DE MEMORIE AL UNEI MATRICE MULTIDIMENSIONALE

C/C++ 472

În secțiunea 457 ați învățat că programele dumneavoastră pot determina volumul de memorie pe care îl consumă o matrice prin înmulțirea numărului de elemente ale matricei cu numărul de octeți ceruți pentru reprezentarea tipului de matrice (cum ar fi 2 pentru *int*, 4 pentru *float* și așa mai departe). Pentru a determina memoria consumată de o matrice multidimensională, puteți efectua același calcul. Pentru determinarea numărului de elemente dintr-o matrice multidimensională, înmulțiți, pur și simplu, numărul rândurilor cu numărul coloanelor. Următoarea expresie ilustrează volumul de memorie pe care îl consumă diferite declarări de matrice:

```
int a[5][10];          // 2*5*10 == 100 octeti
float b[5][8];         // 4*5*8 == 160 octeti
int c[3][4][5];        // 2*3*4*5 == 120 octeti
```

Următorul program, *md_dim.c*, utilizează operatorul *sizeof* pentru a determina numărul de octeți pe care îl consumă diferite declarări de matrice:

```
#include <stdio.h>
void main(void)
  {
    int cutie[3][3];
    float vanzari_an[52][5];
    char pagini[40][60][20];

    printf("Octeti pentru pastrarea int cutie[3][3]
      %d octeti\n", sizeof(cutie));
    printf("Octeti pentru pastrarea float vanzari_an[52][5]
      %d octeti\n", sizeof(vanzari_an));
    printf("Octeti pentru pastrarea char pagini[40][60][20]
      %ld octeti\n", sizeof(pagini));
  }
```

Atunci când compilați și executați programul *md_dim.c*, ecranul dumneavoastră va afișa următoarele:

```
Octeti pentru pastrarea int cutie[3][3] 18 octeti
Octeti pentru pastrarea float vanzari_an[52][5] 1040 octeti
Octeti pentru pastrarea char pagini[40][60][20] 48000 octeti
C:\>
```

473 CICLAREA PRINTR-O MATRICE BIDIMENSIONALĂ



În secțiunea 458 ați învățat cum se utilizează o variabilă pentru a accesa elementele unei matrice. Când programele dumneavoastră lucrează cu matrice bidimensionale, veți utiliza de regulă două variabile pentru a accesa elementele matricei. Următorul program, *tab_2d.c*, utilizează variabilele *rand* și *coloana* pentru a afișa valorile conținute în cadrul matricei *tabel*:

```
#include <stdio.h>

void main(void)
  {
    int rand, coloana;
    float tabel[3][5] = {{1.0, 2.0, 3.0, 4.0, 5.0},
                         {6.0, 7.0, 8.0, 9.0, 10.0},
                         {11.0, 12.0, 13.0, 14.0, 15.0}};

    for (rand = 0; rand < 3; rand++)
      for (coloana = 0; coloana < 5; coloana++)
        printf("tabel[%d][%d] = %f\n", rand, coloana,
          tabel[rand][coloana]);
  }
```

Prin imbricarea buclei *for*, așa cum am arătat, programul va afișa elementele conținute în primul rând al matricei (de la 1.0 la 5.0). După aceea, programul va trece la rândul al doilea, apoi la al treilea rând, afișând fiecare element în cadrul fiecărui rând, unul după altul.

474 TRAVERSAREA PRINTR-O MATRICE TRIDIMENSIONALĂ



În secțiunea 473 ați învățat cum se traversează printr-o matrice bidimensională, utilizând două variabile numite *rand* și *coloana*. Următorul program, *tab_3d.c*, utilizează variabilele *rand, coloana* și *tabel* pentru a traversa printr-o matrice tridimensională:

```
#include <stdio.h>

void main(void)
  {
    int rand, coloana, tabel;
    float val[2][3][5] = {
                         {{1.0, 2.0, 3.0, 4.0, 5.0},
                         {6.0, 7.0, 8.0, 9.0, 10.0},
                         {11.0, 12.0, 13.0, 14.0, 15.0}},

                         {{16.0, 17.0, 18.0, 19.0, 20.0},
                         {21.0, 22.0, 23.0, 24.0, 25.0},
                         {26.0, 27.0, 28.0, 29.0, 30.0}}
                        };

    for (rand = 0; rand < 2; rand++)
```

```
    for (coloana = 0; coloana < 3; coloana++)
      for (tabel = 0; tabel < 5; tabel++)
        printf("val[%d][%d][%d] = %f\n", rand, coloana, tabel,
      val[rand][coloana][tabel]);
}
```

INIȚIALIZAREA UNEI MATRICE MULTIDIMENSIONALE

C/C++ 475

În secțiunea 474 ați învățat cum se afișează conținutul unei matrice tridimensionale utilizân trei variabile: *rand*, *coloana* și *tabel*. Programul *tab_3d.c*, prezentat în secțiunea 474 inițializează matricea tridimensională, așa ca mai jos:

```
float val[2][3][5] = {
                      {{1.0, 2.0, 3.0, 4.0, 5.0},
                      {6.0, 7.0, 8.0, 9.0, 10.0},
                      {11.0, 12.0, 13.0, 14.0, 15.0}},
                      {{16.0, 17.0, 18.0, 19.0, 20.0},
                      {21.0, 22.0, 23.0, 24.0, 25.0},
                      {26.0, 27.0, 28.0, 29.0, 30.0}}
                     };
```

La o primă privire, inițializarea unei matrice multidimensionale poate părea confuză. Pentru înțelege mai bine cum se inițializează o astfel de matrice, acest capitol prezintă câteva exemple d inițializare. Atunci când studiați aceste inițializări, executați inițializarea de la dreapta la stânga:

```
int a[1][2][3] =  {
                    { {1, 2, 3}, {4, 5, 6}
                  }; // Acoladele matricei
int b[2][3][4] =  {
                  { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
                  { {13, 14, 15, 16}, {17, 18, 19, 20},
                    {21, 22, 23, 24}}
                  }; // Acoladele matricei
int c[3][2][4] =  {
                  { {1, 2, 3, 4}, {5, 6, 7, 8}},
                  { {9, 10, 11, 12}, {13, 14, 15, 16}},
                  { {17, 18, 19, 20}, {21, 22, 23, 24}}
                  }; // Acoladele matricei
int d[1][2][3][4] = {
                  { {{1, 2, 3, 4}, {5, 6, 7, 8},
                    {9, 10, 11, 12}},
                  { {13, 14, 15, 16}, {17, 18, 19, 20},
                    {21, 22, 23, 24}}}
                  }; // Acoladele matricei
```

Fiecare inițializare a unei matrice este redată între acolade exterioare. Între cele două acolade exterioare, fiecare dintre elementele matricei sunt definite între alte acolade.

476 TRANSMITEREA UNEI MATRICE BIDIMENSIONALE UNEI FUNCȚII



Când programele dumneavoastră lucrează cu matrice multidimensionale, se poate ivi necesitatea scrierii unor funcții care lucrează cu matrice. În secțiunea 461 ați învățat că atunci când transmiteți o matrice unei funcții, nu trebuie să specificați numărul de elemente ale matricei. Atunci când lucrați cu matrice bidimensionale, nu aveți nevoie să specificați numărul de rânduri ale matricei, dar trebuie să specificați numărul de coloane. Următorul program, *funct_2d.c*, utilizează funcția *un_tab_2d* pentru a afișa conținutul mai multor matrice bidimensionale:

```
#include <stdio.h>

void un_tab_2d (int tab[][10], int rand)
  {
    int i, j;
    for (i = 0; i < rand; i++)
      for (j = 0; j < 10; j++)
        printf("tab[%d][%d] = %d\n", i, j, tab [i][j]);
  }

void main(void)
  {
    int a[1][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}};
    int b[2][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}};
    int c[3][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
                    {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};

    un_tab_2d (a, 1);
    un_tab_2d (b, 2);
    un_tab_2d (c, 3);
  }
```

477 TRATAREA MATRICELOR MULTIDIMENSIONALE CA MATRICE UNIDIMENSIONALE



În secțiunea 476 ați învățat că atunci când transmiteți o matrice bidimensională unei funcții și doriți să accesați poziția rândurilor și a coloanelor matricei, trebuie să specificați numărul de coloane, cum arătăm în continuare:

```
void un_tab_2d (int tab[][10], int rand)
```

Dacă doriţi să lucraţi cu elementele unei matrice multidimensionale, dar nu aveţi nevoie să accesaţi elementele în poziţia lor pe rânduri sau coloane, funcţiile dumneavoastră pot trata matricea multidimensională ca şi când ar fi unidimensională. Următorul program, *sum_2d.c*, returnează suma valorilor unei matrice bidimensionale:

```
#include <stdio.h>

long sum_tab(int tab[], int elemente)
  {
    long sum = 0;
    int i;

    for (i = 0; i < elemente; i++)
      sum += tab[i];
    return(sum);
  }

void main(void)
  {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[2][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}};
    int c[3][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
                    {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};

    printf("Suma elementelor primei matrice %d\n", sum_tab(a, 10));
    printf("Suma elementelor celei de a doua matrice %d\n",
      sum_tab(b, 20));
    printf("Suma elementelor celei de a treia matrice %d\n",
      sum_tab(c, 30));
  }
```

După cum puteţi vedea, funcţia *sum_tab* acceptă matrice uni-, bi- sau multidimensionale. Pentru a înţelege mai bine modul în care lucrează funcţia *sum_tab*, trebuie mai întâi să înţelegeţi cum sunt stocate în memorie, de către compilatorul de C, matricele multidimensionale. Secţiunea 478 va aborda în detaliu modalitatea de stocare a matricelor multidimensionale.

SĂ ÎNŢELEGEM CUM STOCHEAZĂ COMPILATORUL DE C O MATRICE MULTIDIMENSIONALĂ — C/C++ 478

În secţiunea 454 aţi învăţat că atunci când declaraţi o matrice, cum ar fi *int punctaje[100]*, compilatorul de C alocă suficientă memorie pentru a păstra fiecare element al matricei. Atunci când alocaţi o matrice multidimensională, se întâmplă acelaşi lucru. Chiar dacă o matrice multidimensională constă, teoretic, în rânduri, coloane şi pagini, pentru compilator o matrice multidimensională este un interval lung de octeţi. De exemplu, presupunem că programul dumneavoastră declară următoarea matrice:

```
int tabel[3][5];
```

Figura 478 ilustrează imaginea teoretică a unei matrice și memoria reală pe care o utilizează.

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Conceptual

0,0
0,1
0,2
0,3
0,4
1,0
1,1
2,0
2,1
2,2
2,3
2,4

Actual

Figura 478 *Maparea unei matrice multidimensionale în memorie.*

În secțiunea 477 ați creat o funcția care tratează matricele multidimensionale ca unidimensionale pentru a aduna valorile conținute de matrice. Deoarece compilatorul de C mapează matricele multidimensionale la un interval unidimensional de memorie, tratarea matricelor multidimensionale ca unidimensionale este validă.

479 ORDONAREA PE RÂNDURI ȘI ORDONAREA PE COLOANE



În secțiunea 478 ați învățat modul în care compilatorul de C mapează matricele multidimensionale la memoria unidimensională. Atunci când compilatorul mapează în memorie o matrice multidimensională, el are două opțiuni. Așa cum se arată în figura 479, compilatorul poate să plaseze elementele din rândurile matricei înaintea valorilor coloanelor, sau el poate plasa mai întâi elementele coloanelor.

0,0	0,1	0,2
1,0	1,1	1,2

Linie Major	Coloană Major
0,0	0,0
0,1	1,0
0,2	0,1
1,0	1,1
1,1	0,2
1,2	1,2

Figura 479 *Maparea elementelor matricei în memorie.*

Când compilatorul plasează în memorie elementele rândurilor matricei înaintea elementelor coloanelor, compilatorul execută o ordonare *pe rânduri*. La fel, atunci când compilatorul plasează mai întâi elementele din coloane, el execută o ordonare *pe coloane*. Compilatorul de C stochează matricele multidimensionale în ordinea *rândurilor*.

TABLOURI DE STRUCTURI DE MATRICE

C/C++ 480

Tablourile şi structurile vă permit gruparea informaţiilor înrudite. Aşa cum aţi învăţat, compilatorul de C vă permite să creaţi matrice de structuri sau utilizarea matricelor ca membre ale unor structuri. În general, compilatorul de C nu fixează o limită a adâncimii până la care programele dumneavoastră pot ajunge în ceea ce priveşte imbricarea structurilor de date. De exemplu, următoarea declaraţie creează o matrice de 100 structuri de angajaţi. În cadrul fiecărei structuri, există o matrice de structuri *Date* care corespund datei de angajare a salariatului, a primei şi a ultimei sale examinări:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
    struct Date
      {
        int luna;
        int ziua;
        int anul;
      } date_angaj[3];
  } personal[100];
```

Pentru accesarea membrilor şi elementelor matricei, veţi lucra de la stânga la dreapta, începând din exterior, spre interior. De exemplu, următoarea instrucţiune atribuie data de salari a unui angajat:

```
personal[10].date_angaj[0].luna = 7;
personal[10].date_angaj[0].ziua = 7;
personal[10].date_angaj[0].anul = 7;
```

Deşi imbricarea structurilor şi matricelor în această manieră poate fi foarte convenabilă reţineţi că, pe măsură ce programele dumneavoastră imbrică mai multe date şi structuri structurile vor deveni mai dificil de înţeles pentru alţi programatori.

UNIUNEA (UNION)

C/C++ 481

Aşa cum aţi învăţat, structurile permit programelor dumneavoastră să păstreze informaţi legate între ele. În funcţie de scopul programelor dumneavoastră, uneori veţi avea de stoca într-o structură informaţii care nu vor fi numai o valoare sau două. De exemplu, să presupunem că programul dumneavoastră urmăreşte valorile a două date speciale pentru fiecar angajat. Pentru cei angajaţi în prezent, programul urmăreşte numărul de zile lucrate d angajat. Pentru aceia care nu mai lucrează pentru societate, programul urmăreşte ultima dat de angajare. O cale pentru a urmări o astfel de informaţie este utilizarea unei structuri, cu se arată în continuare:

```
struct AngajDate
   {
     int zile_lucru;
     struct Ultima_Data;
       {
         int luna;
         int ziua;
         int anul;
       } ultima_zi;
   };
```

Deoarece programul va utiliza fie membrul *zile_lucru*, fie membrul *ultima_zi*, memoria care păstrează valoarea neutilizată se va irosi. Ca alternativă, compilatorul de C permite programelor dumneavoastră să utilizeze *union*, care alocă numai memoria cerută de cel mai mare dintre membrii ei, cum arătăm în continuare:

```
union AngajDate
   {
     int zile_lucru;
     struct Ultima_Data;
       {
         int luna;
         int ziua;
         int anul;
       } ultima_zi;
   };
```

Pentru a accesa membrii uniunii, folosiți operatorul *dot* la fel ca în cazul structurii. Spre deosebire de structură, însă, uniunea poate să păstreze numai valoarea unui membru. Figura 481 ilustrează modul în care compilatorul de C alocă memorie pentru structură și uniune.



Figura 481 *Alocarea memoriei pentru o structură și o uniune similare.*

Așa cum ați învățat, utilizând uniunile, nu numai că economisiți memorie, dar oferiți programelor dumneavoastră și posibilitatea de a interpreta diferit valorile de memorie.

482 *ECONOMISIREA MEMORIEI CU AJUTORUL UNIUNILOR*



În secțiunea 481 ați învățat că în C puteți stoca informații în cadrul uniunilor. Când utilizați o uniune, compilatorul de C alocă un volum de memorie cerut pentru a păstra membrul cel

mai mare al uniunii. Următorul program, *dimuniun.c*, utilizează operatorul *sizeof* pentru a afișa volumul de memorie pe care îl consumă diferite uniuni:

```
#include <stdio.h>

void main(void)
  {
    union AngajDate
      {
        int zile_lucru;
        struct Date
          {
            int luna;
            int ziua;
            int anul;
          } ultima_zi;
      } angaj_info;

    union Numere
      {
        int a;
        float b;
        long c;
        double d; // Cel mai mare--cere 8 octeti
      } val;

    printf("Dimensiunea AngajDate %d octeti\n", sizeof(angaj_info));
    printf("Dimensiunea Numere %d octeti\n", sizeof(val));
  }
```

Atunci când compilați și executați programul *dimuniun.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Dimensiunea AngajDate 6 octeti
Dimensiunea Numere 8 octeti
C:\>
```

UTILIZAREA REGS – O UNIUNE BINECUNOSCUTĂ C/C++ 483

Așa cum ați învățat, uniunile permit programelor dumneavoastră să reducă solicitările de memorie și să vadă în alt mod informațiile. În secțiunea despre DOS și BIOS a acestei cărți, veți învăța că pentru a accesa serviciile DOS și BIOS, programele dumneavoastră atribuie de obicei parametri (la nivelul limbajului de asamblare) anumitor regiștri ai PC-ului. Pentru ca serviciile DOS și BIOS să fie disponibile programelor dumneavoastră în C, majoritatea compilatoarelor de C dispun de accesarea prin intermediul rutinelor bibliotecii run-time care utilizează uniuni de tip *REGS*:

```
struct WORDREGS
  {
    unsigned int ax, bx, cx, dx, si, di, cflag, flags;
  };
struct BYTEREGS
```

```
    {
      unsigned char al, ah, bl, bh, cl, ch, dl, dh;
    };
union REGS
    {
      struct WORDREGS x;
      struct BYTEREGS h;
    };
```

Atunci când programele dumneavoastră accesează unul dintre regiştrii de uz general ai PC-ului (AX, BX, CX şi DX), PC-ul vă permite să faceţi referinţă la regiştrii de format 16-biţi (cuvânt). Ca alternativă, puteţi să faceţi referinţă la octeţii superiori şi inferiori ai regiştrilor (AL, AH, BL, BH, CL, CH, DL şi DH). Deoarece ambele metode fac referinţă la aceiaşi regiştri, aveţi două căi de accesare aceeaşi locaţie de stocare. Utilizând uniunile, programele dumneavoastră au două căi de accesare a regiştrilor de uz general. Figura 483 ilustrează modul în care compilatorul de C stochează variabilele uniunii REGS în memorie.

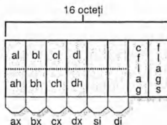


Figura 483 *Modul în care compilatorul de C stochează variabilele uniunii* ***REGS.***

484 UTILIZAREA UNIUNII REGS



În secţiunea 483 aţi învăţat că una dintre cele mai frecvente uniuni utilizate, în programele sub DOS, este uniunea *REGS.* Următorul program, *versx.c,* utilizează uniunea *REGS* pentru a afişa versiunea curentă de DOS, accesând regiştrii generali în forma lor de cuvânt:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    union REGS inregs, outregs;
    inregs.x.ax = 0x3000;
    intdos(&inregs, &outregs);
    printf("Versiunea curenta %d.%d\n", outregs.x.ax & 0xFF,
      outregs.x.ax >> 8);
  }
```

Următorul program, *versb.c,* utilizează regiştrii de octeţi ai uniunii pentru a afişa versiunea curentă de DOS:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    union REGS inregs, outregs;
    inregs.h.ah = 0x30;
    inregs.h.al = 0;
    intdos(&inregs, &outregs);
    printf("Versiunea curenta %d.%d\n", outregs.h.al, outregs.h.ah);
  }
```

STRUCTURILE CÂMP DE BIȚI

C/C++ 485

Multe dintre funcțiile din această carte reduc numărul variabilelor (și, de aici, volumul de memorie alocată) pe care programele dumneavoastră trebuie să le folosească prin returnarea de valori ai căror biți au o semnificație specifică. Când biții care compun valoarea au o semnificație specifică, programele dumneavoastră pot utiliza operatorii C pe biți pentru a extrage valorile (bitul specific). Să presupunem, de exemplu, că programul dumneavoastră trebuie să urmărească 100000 de date calendaristice. Puteți crea o structură de tip *Date* pentru a urmări datele, cum arătăm în continuare:

```
struct Date
    {
      int luna; // de la 1 la 12
      int ziua; // de la 1 la 31
      int anul; // ultimele doua cifre
    };
```

Ca o alternativă, programele dumneavoastră pot utiliza biții specifici în cadrul unei valori *unsigned int* pentru a păstra câmpul dată, cum arătăm în figura 485.

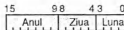


Figura 485 *Utilizarea biților pentru a reprezenta date.*

Apoi, de fiecare dată când programul dumneavoastră trebuie să atribuie data, el poate executa operațiile corecte pe biți, cum arătăm în continuare:

```
unsigned date;

date = luna;
date = date | (ziua << 4);
date = date | (anul << 9);

printf("Luna %d Ziua %d Anul %d\n", date & 0xF, (date >> 4) &
  0x1F, (date >> 9));
```

Pentru a face programele dumneavoastră mai ușor de înțeles compilatorul de C vă permite să creați o *structură câmp de biți*. Atunci când declarați o structură câmp de biți, definiți o structură care specifică înțelesul biților corespunzători:

```
struct Date
     {
       unsigned luna:4;
       unsigned ziua:5;
       unsigned anul:7;
     }date;
```

Programele dumneavoastră vor face apoi referire individuală la câmpul de biți, cum arătăm în continuare:

```
date.luna = 12;
date.ziua = 31;
date.anul = 94;

printf("Luna %d Ziua %d Anul %d\n", date.luna, date.ziua,
   date.anul);
```

Observație: *Atunci când declarați o structură câmp de biți, fiecare dintre membrii structurii trebuie să fie valori de tip* ***unsigned int.***

486 VIZUALIZAREA UNEI STRUCTURI CÂMP DE BIȚI

C/C++

În secțiunea 485 ați învățat că puteți să reprezentați biții în cadrul unei valori, utilizând o structură de tip câmp de biți. Când declarați o structură câmp de biți, compilatorul de C alocă suficienți octeți de memorie pentru a păstra biții structurii. Dacă structura nu utilizează toți biții din ultimul octet, cele mai multe compilatoare de C vor inițializa biții la 0. Pentru a vă ajuta să vizualizați mai bine modul în care compilatorul de C păstrează o structură câmp de biți, figura 486 ilustrează modul în care compilatorul va reprezenta structura câmp de biți *Date*, cum arătăm în următorul cod:

```
struct Date
     {
       unsigned luna:4;
       unsigned ziua:5;
       unsigned anul:7;
     }date;
```

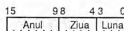


Figura 486 *Modul în care compilatorul de C reprezintă structura câmp de biți* ***Date.***

INTERVALUL DE VALORI AL STRUCTURILOR PE BIȚI

C/C++487

În secțiunea 486 ați învățat că în C puteți reprezenta biții în cadrul unei valori utilizând structura câmp de biți. Atunci când creați o structură câmp de biți, trebuie să alocați suficienți biți pentru a păstra valoarea dorită a fiecărui membru. Pentru a vă ajuta să determinați numărul de biți ceruți, tabelul 487 specifică intervalul de valori pe care un număr dat de biți le poate reprezenta.

Dimensiunea câmpului	Intervalul de valori
1	0-1
2	0-3
3	0-7
4	0-15
5	0-31
6	0-63
7	0-127
8	0-255
9	0-511
10	0-1023
11	0-2047
12	0-4095
13	0-8191
14	0-16383
15	0-32767
16	0-65535

Tabelul 487 *Intervalul de valori pe care programul dumneavoastră le poate reprezenta cu un număr dat de biți.*

CĂUTAREA UNEI VALORI SPECIFICATE ÎNTR-O MATRICE

C/C++488

Așa cum ați învățat, matricele vă permit să păstrați valori corelate de același tip. Puteți să căutați o anumită valoare într-o matrice. Există două modalități uzuale de a căuta într-o matrice: *căutare secvențială* și *căutare binară*. Pentru a executa o căutare secvențială, programul dumneavoastră pornește de la primul element al matricei și caută element cu element, până când programul găsește valoarea dorită sau până când programul ajunge la ultimul element al matricei. De exemplu, următoarea buclă *while* ilustrează modalitatea în care programele dumneavoastră pot căuta valoarea 1500 într-o matrice:

```
gasit = 0;
i = 0;

while ((i < ELEMENTE_TABLOU) && (! gasit))
```

```
    if (matrice[i] == 1500)
      gasit = true;
    else
      i++;
if (i < ELEMENTE_TABLOU)
    printf("Valoarea gasita la elementul %d\n", i);
else
    printf("valoarea nu se gaseste \n");
```

Dacă ați sortat anterior valorile din matrice, de la cea mai mică la cea mai mare, programul dumneavoastră poate executa o căutare binară, despre care veți afla mai multe în secțiunea 489.

489 CĂUTAREA BINARĂ



Așa cum ați învățat, o cale de a localiza o valoare în cadrul unei matrice este căutarea prin toate elementele. Deși o astfel de căutare secvențială este acceptabilă atunci când dimensiunea matricei este mică, ciclarea printr-o matrice mare poate dura mult timp. Dacă programul dumneavoastră a sortat deja valorile din matrice, de la cea mai mică la cea mai mare, el poate utiliza *căutarea binară* pentru a localiza valoarea. Acest tip de căutare este numit căutare binară pentru că la fiecare operație de căutare, se împarte la doi numărul de valori care trebuie examinate.

Cea mai bună cale de a înțelege căutarea binară este asemănarea ei cu o căutare a unui cuvânt în dicționar. Presupunem că vreți să găsiți cuvântul „dalmațian". La început, poate că veți deschide dicționarul la mijlocul lui și veți privi cuvintele de pe pagină. Presupunând că ați deschis la litera M, veți ști că „dalmațian" apare înaintea paginii curente, astfel că ați eliminat deja mai mult de jumătate din cuvintele din dicționar. Dacă deschideți acum la jumătatea paginilor rămase, veți găsi, probabil, cuvintele care încep cu F. Din nou, veți putea elimina o jumătate din alegerile posibile și veți continua căutarea în paginile care preced pagina curentă. De această dată, când veți deschide la jumătatea paginilor rămase, probabil că veți găsi litera C. Cuvântul „dalmațian" apare undeva în paginile dintre C și F. Atunci când veți selecta jumătatea acestor pagini, este posibil să găsiți cuvintele cu D. Prin eliminări repetate de pagini și selectarea paginii din mijloc, veți putea să vă apropiați repede de pagina care conține cuvântul „dalmațian".

Observație: *Pentru a executa o căutare binară, programul dumneavoastră trebuie să sorteze valorile din matrice fie de la cea mai mică la cea mai mare, fie de la cea mai mare la cea mai mică, înainte de începerea căutării.*

490 UTILIZAREA CĂUTĂRII BINARE



Așa cum ați învățat în secțiunea 489, o căutare binară oferă o cale rapidă de căutare a unei anumite valori într-o matrice sortată. Următorul program, *binar.c*, utilizează căutarea binară pentru a căuta mai multe valori în matricea *contor* care conține valori de la 1 la 100. Pentru a vă ajuta să înțelegeți mai bine procedura executată de căutarea binară, funcția *caut_binar* va tipări mesajele care descriu funcționarea ei:

```
#include <stdio.h>
int caut_binar(int tablou[], int val, int dim)
```

```
  {
    int gasit = 0;
    int max = dim, min = 0, med;
    med = (max + min) / 2;
    printf("\n\nCauta %d\n", val);
    while ((! gasit) && (max>= min))
      {
        printf("Minim %d Medie %d Maxim %d\n", min, med, max);
        if (val == tablou[med])
          gasit = 1;
        else if (val < tablou [med])
          max = med - 1;
        else
          min = med + 1;
        med = (max + min) / 2;
      }
    return((gasit) ? med: -1);
  }
void main(void)
  {
    int tablou[100], i;
    for (i = 0; i < 100; i++)
      tablou[i] = i;
    printf("Rezultatul cautarii %d\n", caut_binar(tablou,
       33, 100));
    printf("Rezultatul cautarii %d\n", caut_binar(tablou, 75,
       100));
    printf("Rezultatul cautarii %d\n", caut_binar(tablou, 1,
       100));
    printf("Rezultatul cautarii %d\n", caut_binar(tablou,
       1001, 100));
  }
```

Compilați și executați programul *binar.c* și observați numărul de operații pe care căutarea trebuie să le efectueze pentru a găsi fiecare valoare. Programul utilizează variabilele *max*, *min* și *med* pentru a identifica intervalul de valori în care efectuează căutarea curentă.

SORTAREA UNEI MATRICE

C/C++491

Așa cum ați învățat, matricele vă permit să păstrați valori corelate de același tip. Cum programele dumneavoastră lucrează cu matrice, uneori programele dumneavoastră vor trebui să sorteze valorile unei matrice, fie de la cea mai mică la cea mai mare (ordine ascendentă), fie de la cea mai mare la cea mai mică (ordine descendentă). Programele dumneavoastră pot utiliza câțiva algoritmi diferiți de sortare pentru a sorta matricea, inclusiv *sortarea prin metoda bulelor (bubble)*, *sortarea selectivă*, *sortarea Shell* și *sortarea rapida*. Câteva dintre secțiunile care urmează abordează fiecare dintre aceste metode.

492 SORTAREA PRIN METODA BULELOR

C/C++

Algoritmul de sortare *prin metoda bulelor* este o tehnică simplă de sortare a matricelor care este, de obicei, prima metodă de sortare pe care o învaţă cei mai mulţi programatori. Datorită simplităţii sale, sortarea *prin metoda bulelor* nu este foarte eficientă şi ia mai mult timp de procesare decât oricare altă metodă de sortare. Dacă însă aveţi de sortat matrice de mici dimensiuni, cu 30 de elemente sau mai puţine, utilizarea sortării *prin metoda bulelor* este potrivită. Presupunând că sortaţi valorile de la cea mai mică la cea mai mare, sortarea *prin metoda bulelor* ciclează printre valorile matricei, comparând şi deplasând valorile cele mai mari ale matricei în partea de sus (cum ies bulele la suprafaţa apei). Figura 492 ilustrează patru iteraţii ale sortării *prin metoda bulelor*.

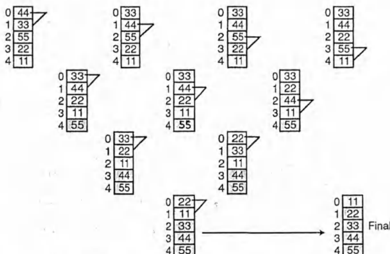


Figura 492 *Patru iteraţii ale sortării prin metoda bulelor.*

Prima iteraţie deplasează valoarea cea mai mare a matricei în partea de sus a matricei. A două iteraţie deplasează a doua valoare ca mărime, în a doua poziţie de sus. A treia iteraţie deplasează cea de a treia valoare în ordinea mărimii, şi aşa mai departe.

493 UTILIZAREA SORTĂRII PRIN METODA BULELOR

C/C++

Secţiunea 492 ilustrează pe scurt funcţionarea *sortării prin metoda bulelor*. Următorul program, *bubble.c*, utilizează sortarea *prin metoda bulelor* pentru a sorta o matrice care conţine 30 de valori aleatoare:

```
#include <stdio.h>
#include <stdlib.h>

void sort_bubble(int tablou[], int dim)
  {
```

```
    int temp, i, j;
    for (i = 0; i < dim; i++)
      for (j = 0; j < dim; j++)
        if (tablou[i] < tablou[j])
          {
            temp = tablou[i];
            tablou[i] = tablou[j];
            tablou[j] = temp;
          }
  }
void main(void)
  {
    int val[30], i;

    for (i = 0; i < 30; i++)
      val[i] = rand() % 100;
    sort_bubble(val, 30);
    for (i = 0; i < 30; i++)
      printf("%d ", val[i]);
  }
```

Observație: *Funcția* ***sort_bubble*** *sortează valori de la cea mai mică la cea mai mare. Pentru a inversa ordinea, modificați operația de comparare astfel:* ***if (tablou[i] > tablou[j])***

SORTAREA SELECTIVĂ

C/C++ 494

Sortarea selectivă este un algoritm simplu de sortare similar cu *sortarea prin metoda bulelor* prezentată în secțiunea 492. La fel ca sortarea *prin metoda bulelor*, programele dumneavoastră trebuie să utilizeze sortarea selectivă numai pentru matricele mici (de 30 sau mai puține elemente). Sortarea selectivă începe prin selectarea unui element al matricei (ca, de exemplu, primul element). Sortarea caută apoi în întreaga matrice până găsește valoarea minimă. Sortarea plasează valoarea minimă în primul element, selectează al doilea element și caută cea de a doua valoare minimă din cele rămase. Figura 494 ilustrează două iterații ale sortării selective asupra unei matrice de valori.

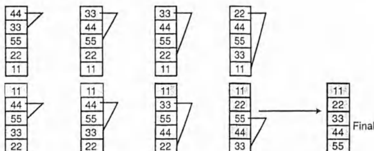


Figura 494 *Sortarea valorilor cu metoda selectivă.*

495 UTILIZAREA SORTĂRII SELECTIVE



Secțiunea 494 a ilustrat pe scurt funcționarea sortării selective. Următorul program, *select.c*, utilizează selecția selectivă pentru a sorta o matrice care conține 30 de valori aleatoare:

```
#include <stdio.h>
#include <stdlib.h>

void sort_select(int tablou[], int dim)
  {
    int temp, curent, j;
    for (curent = 0; curent < dim; curent++)
      for (j = curent + 1; j < dim; j++)
        if (tablou[curent] > tablou[j])
          {
            temp = tablou[curent];
            tablou[curent] = tablou[j];
            tablou[j] = temp;
          }
  }

void main(void)
  {
    int val[30], i;
    for (i = 0; i < 30; i++)
      val[i] = rand() % 100;
    sort_select(val, 30);
    for (i = 0; i < 30; i++)
      printf("%d ", val[i]);
  }
```

Observație: *Funcția **sort_select** sortează valorile de la cea mai mică la cea mai mare. Pentru a inversa ordinea de sortare, modificați comparația astfel: **if(matrice[curent] < matrice[j])**.*

496 SORTAREA SHELL



Sortarea Shell este numită așa după creatorul său, Donald Shell. Tehnica Shell de sortare compară elementele matricei separate printr-o distanță specifică (*gap*) până când elementele pe care le compară în intervalul curent sunt ordonate. Sortarea Shell împarte apoi intervalul la doi și continuă procesul. Când intervalul este în sfârșit unu și nu mai apare nici o modificare, sortarea Shell își încheie prelucrarea. Figura 496 ilustrează modul în care sortarea Shell poate sorta o matrice.

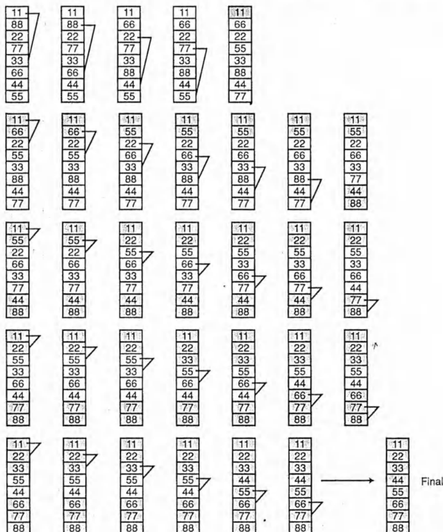


Figura 496 Sortarea unei matrice cu metoda **Shell.**

UTILIZAREA SORTĂRII SHELL

C/C++ 497

Secțiunea 494 a ilustrat pe scurt funcționarea sortării *Shell.* Următorul program, *shell.c,* utilizează sortarea *Shell* pentru a sorta o matrice care conține 50 de valori aleatoare:

```
#include <stdio.h>
#include <stdlib.h>
void sort_shell(int tablou[], int dim)
  {
    int temp, gap, i, modificari;

    gap = dim / 2;
    do
     {
      do
       {
        modificari = 0;
        for (i = 0; i < dim - gap; i++)
          if (tablou[i] > tablou[i + gap])
           {
             temp = tablou[i];
             tablou[i] = tablou[i + gap];
             tablou[i + gap] = temp;
             modificari = 1;
           }
       }
      while (modificari);
     }
    while (gap = gap / 2);
  }
void main(void)
  {
    int val[50], i;
    for (i = 0; i < 50; i++)
      val[i] = rand() % 100;
    sort_shell(val, 50);
    for (i = 0; i < 50; i++)
      printf("%d ", val[i]);
  }
```

Observaţie: *Funcţia* ***sort_shell*** *sortează de la cel mai mic la cel mai mare. Pentru a inversa ordinea de sortare, modificaţi astfel:* ***if (matrice[i] < matrice[i + gap]).***

498 SORTAREA RAPIDĂ



Pe măsură ce numărul de elemente ale matricelor dumneavoastră creşte, *sortarea rapidă* devine una dintre cele mai rapide tehnici de sortare pe care programele dumneavoastră pot să le utilizeze. Sortarea rapidă consideră matricea ca pe o listă de valori. Când începe sortarea, ea selectează valoarea de mijloc a listei ca *separator de listă*. Sortarea împarte apoi lista în două liste, una cu valorile mai mici decât separatorul de listă şi o a doua ale cărei valori sunt mai mari sau egale cu separatorul de listă. Sortarea se invocă apoi recursiv în

ambele liste. La fiecare invocare, ea împarte elementele în liste mai mici. Figura 498 ilustrează cum funcționează sortarea rapidă în cazul unei matrice de 10 valori.

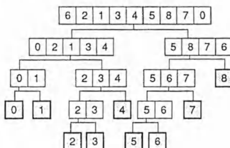


Figura 498 *Sortarea valorilor cu sortarea rapidă.*

UTILIZAREA SORTĂRII RAPIDE

C/C++499

Secțiunea 498 a ilustrat pe scurt funcționarea sortării rapide. Următorul program, *rapid.c*, utilizează sortarea rapidă pentru a sorta o matrice care conține 100 de valori aleatoare:

```
#include <stdio.h>
#include <stdlib.h>

void sort_rapid(int tablou[], int prim, int ultim)
  {
    int temp, min, max, separator_lista;
    min = prim;
    max = ultim;
    separator_lista = tablou[(prim + ultim) / 2];
    do
      {
        while (tablou[min] < separator_lista)
         min++;
        while (tablou[max] > separator_lista)
         max--;
        if (min <= max)
          {
            temp = tablou[min];
            tablou[min++] = tablou[max];
            tablou[max--] = temp;
          }
      }
    while (min <= max);
    if (prim < max)
     sort_rapid(tablou, prim, max);
    if (min < ultim)
     sort_rapid(tablou, min, ultim);
```

```
}
void main(void)
{
    int val[100], i;

    for (i = 0; i < 100; i++)
      val[i] = rand() % 100;
    sort_rapid(val, 0, 99);
    for (i = 0; i < 100; i++)
      printf("%d ", val[i]);
}
```

Observație: *Funcția* ***sort_rapid*** *sortează valorile de la cea mai mică la cea mai mare. Pentru a inversa ordinea de sortare, modificați în mod corespunzător cele două instrucțiuni* ***while****, cum arătăm în continuare:*

```
while (tablou[min] > separator_lista)
     min++;

while (tablou[max] < separator_lista)
     max--;
```

500 PROBLEME CU SOLUȚIILE DE SORTARE ANTERIOARE

C/C++

Câteva dintre secțiunile anterioare au prezentat diferite tehnici de sortare pe care programele dumneavoastră le pot utiliza pentru a sorta matrice. Fiecare dintre secțiunile prezentate, însă, lucrează cu matrice de tip *int*. Dacă programele dumneavoastră necesită sortarea unor alte tipuri de matrice, trebuie să creați funcții noi. De exemplu, pentru a sorta o matrice de tip *float*, programele dumneavoastră trebuie să modifice antetul funcției *sort_rapid* și declarările variabilelor:

```
void sort_rapid(float tablou[], int prim, int ultim)
  {
    float temp, separator_lista;
    int min, max;
```

Dacă doriți să sortați o matrice de valori de tip *long*, trebuie să creați o altă funcție. Așa cum ați învățat, programele dumneavoastră pot utiliza funcția *qsort* a bibliotecii run_time de C pentru a sorta diferite tipuri de matrice. Funcția *qsort* utilizează redirectarea memoriei pentru a sorta valori de orice tip.

501 SORTAREA UNEI MATRICE DE ȘIRURI DE CARACTERE

C/C++

Așa cum ați învățat, limbajul C vă permite să creați o matrice de șiruri de caractere, cum arătăm în continuare:

```
char *zile[] = {"Luni", "Marti", "Miercuri"};
```

Cum se întâmplă ca programele dumneavoastră să necesite sortarea matricelor de alte tipuri, același lucru este valabil și pentru sortarea matricelor de șiruri de caractere. Următorul program, *sort_sir.c*, utilizează sortarea *bubble* pentru a sorta o matrice de șiruri de caractere:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void sort_bubble(char *tablou[], int dim)
  {
    char *temp;
    int i, j;

    for (i = 0; i < dim; i++)
      for (j = 0; j < dim; j++)
        if (strcmp(tablou[i], tablou[j]) < 0)
          {
            temp = tablou[i];
            tablou[i] = tablou[j];
            tablou[j] = temp;
          }
  }

void main(void)
  {
    char *val[] = {"AAA", "CCC", "BBB", "EEE", "DDD"};
    int i;
    sort_bubble(val, 5);
    for (i = 0; i < 5; i++)
      printf("%s ", val[i]);
  }
```

Când funcția sortează matricea cu șiruri de caractere, funcția nu modifică conținutul șirului de caractere pentru a rearanja matricea; în schimb, ea aranjează pointerii șirului de caractere astfel încât șirurile de caractere să fie ordonate.

CĂUTAREA ÎNTR-UN ȘIR DE CARACTERE CU FUNCȚIA LFIND

C/C++ 502

Așa cum ați învățat, o operație de căutare secvențială caută elementele dintr-o matrice în ordine, până când găsește valoarea specificată. Pentru a ajuta programele dumneavoastră să caute în orice tip de matrice, biblioteca run-time a limbajului C, vă pune la dispoziție funcția *lfind*:

```
#include <stdlib.h>

void *lfind(const void *element, void *baza,
            size_t *nr_intrari, size_t dim_element,
            int (*compar)(const void*, const void *));
```

După cum vedeţi, funcţia face intens uz de pointeri. Parametrul *element* este un pointer la valoarea dorită. Parametrul *baza* este un pointer la începutul matricei. Parametrul *nr_intrari* este un pointer la numărul de elemente ale matricei. Parametrul *dim_element* specifică numărul de octeţi pe care îi cere fiecare element al matricei. În sfârşit, parametrul *compar* este un pointer la o a doua funcţie care compară două elemente ale matricei. Spre deosebire de funcţiile arătate anterior, care returnau un indice al matricei la valoarea dorită, funcţia *lfind* returnează un pointer la valoarea dorită sau valoarea 0 dacă funcţia *lfind* nu găseşte elementul. Următorul program, *lfind.c*, utilizează funcţia *lfind* pentru a căuta valori de tip *int* şi valori de tip *float*:

```
#include <stdlib.h>
#include <stdio.h>

int compar_int(int *a, int *b)
  {
    return(*a - *b);
  }

int compar_float(float *a, float *b)
  {
    return((*a == *b) ? 0: 1);
  }

void main(void)
  {
    int valori_int[] = {1, 3, 2, 4, 5};
    float valori_float[] = {1.1, 3.3, 2.2, 4.4, 5.5};
    int *ptr_int, val_int = 2, elemente = 5;
    float *ptr_float, val_float = 33.3;

    ptr_int = lfind(&val_int, valori_int, &elemente, sizeof(int),
                    (int (*) (const void *, const void *))
                     compar_int);

    if (*ptr_int)
     printf("Valoarea %d gasita\n", val_int);
    else
    printf("Valoarea %d nu e gasita\n", val_int);

    ptr_float = lfind(&val_float, valori_float, &elemente,
                       sizeof(float), (int (*) (const void *,
                       const void *)) compar_float);

    if (*ptr_float)
     printf("Valoarea %3.1f gasita\n", val_float);
    else
     printf("Valoarea %3.1f nu e gasita\n", val_float);
  }
```

Utilizând pointerii, funcţia este capabilă să elimine conflictele legate de tipuri, care au afectat funcţiile de căutare şi sortare discutate anterior.

CĂUTAREA UNEI VALORI CU FUNCȚIA LSEARCH

C/C++ 503

În secțiunea 502 ați învățat cum se utilizează funcția *lfind* pentru a căuta printre valorile unei matrice, un anumit element. Dacă funcția nu găsește elementul, ea returnează 0. În funcție de programele dumneavoastră, puteți să adăugați un element la matrice, dacă funcția nu îl găsește. Pentru astfel de cazuri, programele dumneavoastră pot utiliza funcția *lsearch*:

```
#include <stdlib.h>

void *lsearch(const void *element, void *baza, size_t *nr_intrari,
    size_t dim_element, int (* compar)(const void*, const
    void *));
```

Următorul program, *lsearch.c*, utilizează funcția *lsearch* pentru a căuta valoarea 1500. Dacă funcția *lsearch* nu găsește valoarea, ea va adăuga valoarea la matrice:

```
#include <stdlib.h>
#include <stdio.h>

int compar_int(int *a, int *b)
  {
    return(*a - *b);
  }

void main(void)
  {
    int valori_int[10] = {1, 3, 2, 4, 5};
    int *ptr_int, val_int = 1500, elemente = 5, i;

    printf("Continutul matricei inainte de cautare\n");
    for (i = 0; i < elemente; i++)
      printf("%d ", valori_int[i]);
    ptr_int = lsearch(&val_int, valori_int, &elemente, sizeof(int),
      (int (*) (const void *, const void *)) compar_int);
    printf("\nContinutul matricei dupa cautare\n");
    for (i = 0; i < elemente; i++)
      printf("%d ", valori_int[i]);
  }
```

După cum puteți vedea, atunci când funcția adaugă valoarea, ea actualizează, de asemenea, valoarea parametrului care specifică numărul elementelor matricei.

Observație: *Atunci când programele dumneavoastră utilizează funcția **lsearch**, trebuie să includeți un spațiu în plus în cadrul matricei, în care puteți adăuga valoarea.*

CĂUTAREA ÎNTR-O MATRICE SORTATĂ CU FUNCȚIA BSEARCH

C/C++ 504

În secțiunea 489 ați învățat că o căutare binară localizează o valoare într-o matrice sortată, prin reducerea repetată a numărului de elemente ale matricei, prin împărțirea la 2 la fiecare

iterație. Pentru a ajuta programele dumneavoastră să efectueze operații de căutare binară, biblioteca run-time a limbajului C dispune de funcția *bsearch*:

```
#include <stdlib.h>

void *bsearch(const void *element, void *baza,
          size_t *nr_elemente, size_t dim,
          int (*compar)(const void*, const void *));
```

La fel ca funcția *lsearch* despre care ați învățat în secțiunea 503, funcția *bsearch* utilizează extrem de mult pointerii. Parametrul *element* este un pointer la valoarea dorită. Parametrul *baza* este un pointer la începutul matricei. Parametrul *nr_elemente* specifică numărul de elemente ale matricei. Parametrul *dim* specifică numărul de octeți ceruți pentru fiecare element al matricei. În sfârșit, parametrul *compar* este un pointer la o a doua funcție care compară două elemente ale matricei. Spre deosebire de funcțiile arătate anterior, care returnau un indice al matricei la valoarea dorită, funcția *bsearch* returnează un pointer la valoarea dorită sau valoarea 0 dacă funcția *bsearch* nu găsește elementul respectiv. Următorul program, *bsearch.c*, utilizează funcția *bsearch* pentru a căuta în două matrice diferite, una cu valori de tip *int* și cealaltă cu valori de tip *float*:

```
#include <stdlib.h>
#include <stdio.h>

int compar_int(int *a, int *b)
  {
    return(*a - *b);
  }
int compar_float(float *a, float *b)
  {
    return((*a == *b) ? 0: 1);
  }

void main(void)
  {
    int valo_int[] = {1, 3, 2, 4, 5};
    float valo_float[] = {1.1, 3.3, 2.2, 4.4, 5.5};

    int *ptr_int, val_int = 2, elemente = 5;
    float *ptr_float, val_float = 33.3;

    ptr_int = bsearch(&valo_int, valo_int, elemente, sizeof(int),
      (int (*) (const void *, const void *)) compar_int);

    if (*ptr_int)
      printf("Valoarea %d gasita\n", val_int);
    else
      printf("Valoarea %d nu e gasita\n", val_int);
    ptr_float = bsearch(&val_float, valo_float, elemente,
                        sizeof(float), (int (*) (const void *,
                        const void *)) compar_float);

    if (*ptr_float)
```

```
    printf("Valoarea %d gasita\n", val_float);
  else
    printf("Valoarea %d nu e gasita\n", val_float);
}
```

Observaţie: *Pentru a utiliza funcţia* ***bsearch****, valorile matricei trebuie să fie sortate de la cea mai mică la cea mai mare.*

SORTAREA UNEI MATRICE CU FUNCŢIA QSORT

C/C++ 505

Secţiunea 498 v-a învăţat că o operaţie de sortare rapidă sortează elementele matricei tratând-o ca o listă. Deoarece sortarea rapidă separă în mod repetat elementele în liste sortate mai mici, ea este foarte eficientă. Pentru a ajuta programele dumneavoastră să sorteze matrice de orice tip, utilizând sortarea rapidă, biblioteca run-time a limbajului C furnizează funcţia *qsort*:

```
#include <stdlib.h>
void *qsort(void *baza, size_t *nr_intrari, size_t
        dim_element,int (*compar)(const void*,
        const void *));
```

La fel ca funcţiile *lsearch* şi *bsearch* despre care aţi învăţat deja, funcţia *qsort* utilizează extrem de mult pointerii. Parametrul *baza* este un pointer la începutul matricei. Parametrul *nr_intrari* specifică numărul de elemente ale matricei. Parametrul *dim_element* specifică numărul de octeţi ceruţi pentru fiecare element al matricei. În sfârşit, parametrul *compar* este un pointer la o a doua funcţie care compară două elemente ale matricei, cum arătăm în continuare:

```
*a < *b    // returneaza o valoare < 0
*a == *b   // returneaza 0
*a > *b    // valoare > 0
```

Următorul program, *qsort.c*, utilizează funcţia *qsort* pentru a căuta valori de tip *int* şi valori de tip *float*:

```
#include <stdlib.h>
#include <stdio.h>

int compar_int(int *a, int *b)
  {
    if (*a < *b)
      return(-1);
    else if (*a == *b)
      return(0);
    else
      return(1);
  }

int compar_float(float *a, float *b)
```

```
  {
    if (*a < *b)
      return(-1);
    else if (*a == *b)
      return(0);
    else
      return(1);
  }
void main(void)
  {
    int valori_int[] = {51, 23, 2, 44, 45};
    float valori_float[] = {21.1, 13.3, 22.2, 34.4, 15.5};
    int elemente = 5, i;
    qsort(valori_int, elemente, sizeof(int), (int (*)
          (const void *, const void *)) compar_int);
    for (i = 0; i < elemente; i++)
      printf("%d ", valori_int[i]);
    putchar('\n');
    qsort(valori_float, elemente, sizeof(float),
          int (*) (const void *, const void *)) compar_float);
    for (i = 0; i < elemente; i++)
      printf("%4.1f ", valori_float[i]);
  }
```

506 DETERMINAREA NUMĂRULUI DE ELEMENTE ALE MATRICEI

C/C++

Câteva dintre secțiunile anterioare au inclus numărul de elemente ale matricei ca un parametrul la funcții. Dacă numărul de elemente dintr-o matrice se poate modifica, puteți reduce numărul de modificări pe care trebuie să le faceți programului dumneavoastră, utilizând o valoare constantă, ca mai jos:

```
#define NUM_ELEMENTE = 5
```

Altfel, programele dumneavoastră pot folosi operatorul *sizeof* pentru a determina numărul de elemente din matrice, cum arătăm în continuare:

```
elemente = sizeof(matrice) / sizeof(tablou[0]);
```

Următorul program, *num_elem.c*, utilizează operatorul *sizeof* pentru a determina numărul de elemente din două matrice și apoi afișează numărul de elemente din acele matrice:

```
#include <stdio.h>

void main(void)
  {
    int valori_int[] = {51, 23, 2, 44, 45};
    float valori_float[] = {21.1, 13.3, 22.2, 34.4, 15.5};
```

```
    printf("Numarul de elemente in valori_int %d\n",
           sizeof(valori_int) / sizeof(valori_int[0]));
    printf("Numarul de elemente in valori_float %d\n",
           sizeof(valori_float) / sizeof(valori_float[0]));
  }
```

SĂ ÎNŢELEGEM POINTERII CA ADRESE

C/C++ 507

Aşa cum aţi învăţat în primul capitol al acestei cărţi, o *variabilă* este numele unei locaţii de memorie care este aptă de a păstra o valoare de un anumit tip. Programul dumneavoastră face referinţă la fiecare locaţie de memorie prin utilizarea unei *adrese* unice. *Pointerul* este o variabilă care conţine o adresă. Limbajul de programare C utilizează foarte frecvent pointerii. Atunci când transmiteţi o matrice sau şiruri de caractere unei funcţii, compilatorul de C transmite un pointer. De asemenea, când o funcţie trebuie să modifice valoarea unui parametru, programul trebuie să transmită funcţiei un pointer la adresa de memorie a variabilei. Câteva dintre secţiunile care urmează abordează în detaliu pointerii.

DETERMINAREA ADRESEI UNEI VARIABILE

C/C++ 508

Pointerul este o adresă la o locaţie de memorie. Atunci când programele dumneavoastră lucrează cu matrice (şi cu şiruri de caractere), programul lucrează cu pointeri la primul element al matricei. Atunci când programele dumneavoastră au nevoie să determine adresa unei variabile, programele trebuie să utilizeze *operatorul adresă* al limbajului C, caracterul *ampersand* (&). De exemplu, următorul program, *adresa.c*, utilizează operatorul *adresă* pentru a afişa adresa câtorva variabile diferite:

```
#include <stdio.h>

void main(void)
  {
    int contor = 1;
    float salariu = 40000.0;
    long distanta = 1234567L;
    printf("Adresa variabilei contor este %x\n", &contor);
    printf("Adresa variabilei salariu este %x\n", &salariu);
    printf("Adresa variabilei distanta este %x\n", &distanta);
  }
```

Atunci când compilaţi şi executaţi programul *adresa.c*, ecranul dumneavoastră va afişa următorul rezultat:

```
Adresa variabilei contor este fff4
Adresa variabilei salariu este fff0
Adresa variabilei distanta este ffec
C:\>
```

509 SĂ ÎNȚELEGEM CUM TRATEAZĂ COMPILATORUL DE C MATRICELE CA POINTERI



Așa cum deja ați învățat, compilatorul de C tratează matricele ca pointeri. Atunci când programul transmite o matrice unei funcții, de exemplu, compilatorul transmite adresa de început a matricei. Următorul program, *adrstab.c*, afișează adresa de început a câtorva matrice diferite:

```
#include <stdio.h>

void main(void)
  {
    int contor[10];
    float salarii[5];
    long distante[10];
    printf("Adresa matricei contor este %x\n", &contor);
    printf("Adresa matricei salarii este %x\n", &salarii);
    printf("Adresa matricei distante este %x\n", &distante);
  }
```

Atunci când compilați și executați programul *adrstab.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Adresa matricei contor este ffe2
Adresa matricei salarii este ffce
Adresa matricei distante este ffa6
C:\>
```

510 APLICAREA OPERATORULUI ADRESĂ (&) UNEI MATRICE



Așa cum ați învățat, compilatorul de C tratează matricele ca pointeri la primul element al matricei. În secțiunea 508, ați învățat că C utilizează operatorul *adresă* (&) pentru a returna adresa variabilei. Dacă aplicați operatorul *adresă* unei matrice, compilatorul de C returnează adresa de început a matricei. De aceea, aplicarea operatorului *adresă* unei matrice este redundantă. Următorul program, *doua_adr.c*, afișează adresa de început a unei matrice, urmată de pointerul pe care operatorul *adresă* al limbajului C îl returnează:

```
#include <stdio.h>

void main(void)
  {
    int contor[10];
    float salarii[5];
    long distante[10];
    printf("Adresa matricei contor este %x &contor este %x\n",
      contor, &contor);
    printf("Adresa matricei salarii este %x &salarii este
      %x\n", salarii, &salarii);
    printf("Adresa matricei distante este %x &distante este
```

```
        %x\n", distante, &distante);
    }
```

Atunci când compilați și executați programul *doua_adr.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Adresa matricei contor este ffe2 &contor este ffe2
Adresa matricei salarii este ffce &salarii este ffce
Adresa matricei distante este ffa6 &distante este ffa6
C:\>
```

DECLARAREA VARIABILELOR POINTERI

C/C++ 511

Pe măsură ce programele dumneavoastră devin mai complexe, veți observa că lucrați cu pointerii în mod obișnuit. Pentru a păstra pointerii, programul dumneavoastră trebuie să declare variabile pointeri. Pentru a declara un pointer, trebuie să specificați tipul valorii pe care pointerul o indică (cum sunt *int, float, char* și așa mai departe) și un asterisc (*) înaintea numelui variabilei. De exemplu, următoarea instrucțiune declară un pointer la o valoare de tip *int*:

```
int *iptr;
```

Ca pentru orice variabilă, trebuie să atribuiți o valoare unei variabile pointer înainte de a putea folosi pointerul în cadrul programului. Atunci când atribuiți o valoare unui pointer, de fapt atribuiți o adresă. Presupunând că ați declarat anterior variabila *int contor*, următoarea instrucțiune atribuie adresa variabilei *contor* pointerului *iptr*:

```
iptr = &contor; // Atribuie adresa lui contor, lui iptr
```

Următorul program, *iptr.c*, declară variabila pointer *iptr* și atribuie pointerului adresa variabilei *contor*. Programul afișează apoi valoarea variabilei pointer, împreună cu adresa variabilei *contor*.

```
#include <stdio.h>

void main(void)
  {
    int *iptr; // Declara variabila pointer
    int contor = 1;

    iptr = &contor;
    printf("Valoarea lui iptr %x Valoarea lui contor %d Adresa
    lui contor %x\n", iptr, contor, &contor);
  }
```

Atunci când compilați și executați programul *iptr.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
Valoarea lui iptr fff2 Valoarea lui contor 1 Adresa lui contor
fff2
C:\>
```

512 DEREFERENȚIEREA UNUI POINTER



Așa cum ați învățat, un pointer conține o adresă care indică spre o valoare de un anumit tip. Utilizând adresa pe care o conține un pointer, puteți determina valoarea din memorie către care indică pointerul. *Dereferențierea unui pointer* este procesul de accesare a valorii la o anumită locație de memorie. Pentru a dereferenția valoarea unui pointer, utilizați operatorul de *redirectare, asteriscul* (*). De exemplu, următoarea instrucțiune *printf* afișează valoarea către care indică pointerul de tip întreg *iptr*:

```
printf("Valoarea indicata de catre iptr este %d\n", *iptr);
```

La fel, instrucțiunea următoare atribuie valoarea la care indică variabila *iptr*, variabilei *contor*:

```
contor = *iptr;
```

În sfârșit, următoarea instrucțiune atribuie valoarea 7 locației de memorie către care indică *iptr*:

```
*iptr = 7;
```

Observație: *Pentru a utiliza valoarea păstrată în locația de memorie indicată de un pointer, trebuie să dereferențiați valoarea pointerului, utilizând operatorul de* ***redirectare, asterisc (*)****.*

513 UTILIZAREA VALORILOR POINTER



În secțiunea 510, ați învățat că puteți atribui o adresă unei variabile pointer, utilizând operatorul *adresă ampersand* (&). În secțiunea 512 ați învățat că pentru a accesa valoarea stocată în memorie, la locația de memorie indicată de un pointer, trebuie să utilizați operatorul *de redirectare asterisc* (*). Următorul program, *ptr_demo.c*, atribuie pointerului *iptr* de tip *int* adresa variabilei *contor*. Programul afișează apoi valoarea pointerului și valoarea stocată la locația pe care pointerul o indică (valoarea lui *contor*). Programul modifică apoi valoarea indicată de către pointer, ca mai jos:

```
#include <stdio.h>

void main(void)
  {
    int contor = 10;
    int *iptr;          // Declara valoarea pointerului
    iptr = &contor;     // Atribuie adresa
    printf("Adresa in iptr %x Valoarea la *iptr %d\n", iptr,
      *iptr);
    *iptr = 25;         // Modifica valoarea in memorie
    printf("Valoarea variabilei contor %d\n", contor);
  }
```

UTILIZAREA POINTERILOR CU PARAMETRII FUNCŢIILOR

C/C++ 514

Capitolul despre funcţii al acestei cărţi examinează în detaliu procesul de transmitere a parametrilor la funcţii. Aşa cum veţi învăţa, atunci când aveţi nevoie să modificaţi valoarea unui parametru, trebuie să transmiteţi funcţiei un pointer la un parametru. Următorul program, *sch_val.c*, utilizează pointerii la doi parametri de tip *int* pentru a modifica valorile variabilelor, cum arătăm în continuare:

```
#include <stdio.h>

void modific_val(int *a, int *b)
   {
     int temp;

     temp = *a; // Pastreaza temporar valoarea indicata de a
     *a = *b;  // Atribuie valoarea lui b lui a
     *b = temp; // Atribuie valoarea lui a lui b
   }

void main(void)
   {
     int unu = 1, doi = 2;
     modific_val(&unu, &doi);
     printf("unu contine %d doi contine %d\n", unu, doi);
   }
```

După cum puteţi vedea, în cadrul funcţiei instrucţiunile dereferenţiază pointerii, utilizând operatorul de *redirectare* (*). Programul transmite fiecare adresă a variabilei la funcţie, utilizând operatorul *adresă* (&).

ARITMETICA POINTERILOR

C/C++ 515

Un pointer este o adresă care indică spre o valoare de un anumit tip din memorie. Mai simplu, un pointer este o valoare care indică o anumită locaţie de memorie. Dacă adăugaţi valoarea 1 unui pointer, pointerul va indica următoarea locaţie din memorie. Dacă adăugaţi 5 la valoarea pointerului, pointerul va indica locaţia de memorie care este cu cinci locaţii după adresa curentă. Însă, aritmetica pointerilor nu este atât de simplă cum credeţi. De exemplu, să presupunem că un pointer conţine adresa 1000. Dacă adunaţi 1 la pointer, vă veţi aştepta ca rezultatul să fie 1001. Adresa rezultată, însă, depinde de tipul pointerului. De exemplu, dacă adăugaţi 1 la o variabilă pointer de tip *char* (care conţine 1000), adresa rezultată este 1001. Dacă adunaţi 1 la un pointer de tip *int* (care cere doi octeţi de memorie), adresa rezultată este 1002. De asemenea, dacă adăugaţi 1 la un pointer de tip *float* (care cere patru octeţi), adresa rezultată este 1004. Atunci când efectuaţi operaţii aritmetice cu pointeri, ţineţi seama de tipul pointerului. În plus faţă de adunarea de valori la pointeri, programele dumneavoastră pot scădea valori sau pot scădea doi pointeri. Câteva dintre secţiunile acestui capitol prezintă diferite operaţii aritmetice cu pointeri.

516 INCREMENTAREA ȘI DECREMENTAREA UNUI POINTER



Atunci când programele dumneavoastră lucrează cu pointeri, una dintre cele mai comune operații pe care ele le efectuează este incrementarea și decrementarea valorii unui pointer pentru a indica următoarea sau anterioara locație de memorie. Următorul program, *ptr_tab.c*, atribuie adresa de început a unei matrice de valori întregi la pointerul *iptr*. Programul incrementează apoi valoarea pointerului pentru a afișa cele cinci elemente conținute de matrice:

```
#include <stdio.h>

void main(void)
  {
    int valori[5] = {1, 2, 3, 4, 5};
    int contor;
    int *iptr;
    iptr = valori;
    for (contor = 0; contor < 5; contor++)
      {
        printf("%d\n", *iptr);
        iptr++;
      }
  }
```

Atunci când compilați și executați programul *ptr_tab.c*, ecranul dumneavoastră va afișa valorile de la 1 la 5. Programul atribuie inițial pointerul la adresa de început a matricei, apoi programul incrementează pointerul pentru a indica fiecare element.

517 COMBINAREA UNEI REFERINȚE LA UN POINTER CU INCREMENTAREA



În secțiunea 516 ați utilizat pointerul *iptr* pentru a afișa conținutul unei matrice. Pentru a afișa conținutul matricei, pointerul utilizează bucla *for*, cum arătăm în continuare:

```
for (contor = 0; contor < 5; contor++)
      {
        printf("%d\n", *iptr);
        iptr++;
      }
```

După cum puteți vedea, bucla *for* accesează valoarea pointerului pe o linie și apoi incrementează pointerul în următoarea. Așa cum ați învățat, puteți utiliza operatorul C *de incrementare postfixat* pentru a utiliza valoarea variabilei și apoi să incrementați valoarea. Următoarea buclă *for* utilizează operatorul de incrementare postfixat pentru a referenția valoarea indicată de variabila pointer și apoi incrementează valoarea pointerului:

```
for (contor = 0; contor < 5; contor++)
   printf("%d\n", *iptr++);
```

CICLAREA PRINTR-UN ȘIR DE CARACTERE UTILIZÂND UN POINTER

C/C++518

Capitolul despre șiruri de caractere al acestei cărți utilizează foarte mult pointerii. Așa cum ați învățat, un șir este o matrice de caractere terminată prin caracterul *NULL*. Următorul program, *ptr_sir.c*, utilizează funcția *vezi_sir* pentru a afișa un șir de caractere utilizând un pointer:

```
#include <stdio.h>

void vezi_sir(char *sir)
  {
    while (*sir)
      putchar(*sir++);
  }

void main(void)
  {
    vezi_sir("Totul despre C/C++");
  }
```

După cum puteți vedea, funcția *vezi_sir* declară variabila *sir* ca pointer. Utilizând pointerii, funcția pur și simplu ciclează prin caracterele șirului, până întâlnește caracterul *NULL*. Pentru a afișa caracterul, funcția *vezi_sir* dereferențiază mai întâi adresa pointerului (obținând caracterul). Apoi, funcția incrementează pointerul pentru a indica următorul caracter al șirului.

UTILIZAREA FUNCȚIILOR CARE RETURNEAZĂ POINTERI

C/C++519

Așa cum ați învățat, funcțiile pot returna o valoare programelor dumneavoastră. Valoarea pe care funcția o returnează este întotdeauna de tipul declarat în prototipul funcției sau în antet (tipuri ca *int*, *float* sau *char*). În plus față de returnarea acestor tipuri de bază, funcțiile pot declara pointeri la valori. De exemplu, funcția *fopen*, pe care o utilizează majoritatea programelor în C pentru a deschide un flux de fișier returnează un pointer la o structură de tip *FILE*, cum arătăm în continuare:

```
FILE *fopen(const char *numecale, const char *mod);
```

În mod similar, multe dintre funcțiile prezentate în capitolul despre șiruri de caractere a acestei cărți returnează pointeri la șiruri de caractere. Dacă analizați prototipurile de funcții prezentate în această carte, observați funcțiile care returnează *un pointer la o valoare*, spre deosebire de valorile cu tipuri de bază.

CREAREA UNEI FUNCȚII CARE RETURNEAZĂ UN POINTER

C/C++520

În secțiunea 519 ați învățat că multe dintre funcțiile bibliotecii C run-time returnează pointeri. Cum programele dumneavoastră devin mai complexe, veți crea funcții care returnează pointeri la un anumit tip. De exemplu, următorul program, *ptr_maj.c*, creează o

funcție numită *sir_majuscule* care convertește fiecare dintre caracterele unui șir în majuscule și apoi returnează un pointer la șir:

```
#include <stdio.h>
#include <ctype.h>

char *sir_majuscule(char *sir)
  {
    char *adresa_start;

    adresa_start = sir;
    while (*sir)
      {
        *sir = toupper(*sir);
        sir++;
      }
      return(adresa_start);
  }

void main(void)
  {
    char *titlu = "Totul despre C/C++";
    char *sir;

    sir = sir_majuscule(titlu);
    printf("%s\n", sir);
    printf("%s\n", sir_majuscule("Matrice si pointeri"));
  }
```

După cum vedeți, pentru a crea o funcție care returnează un pointer, veți plasa un asterisc înaintea numelui funcției, ca mai jos:

```
char *sir_majuscule(char *sir);
```

521 O MATRICE DE POINTERI

C/C++

Câteva dintre secțiunile prezentate în acest capitol prezintă în detaliu matricele. Până acum, toate matricele au utilizat tipurile de bază ale limbajului C (cum sunt *int*, *float* sau *char*); totuși, limbajul C nu restrânge matricele numai la aceste tipuri simple. Așa cum puteți crea funcții care returnează pointeri, puteți, de asemenea, să creați matrice de pointeri. Pentru a stoca șirurile de caractere, veți utiliza cel mai adesea matricele de pointeri. Ca exemplu, următoarea declarare creează o matrice, numită *saptamana*, care conține pointeri la șiruri de caractere:

```
char *saptamana[7] = {"luni", "marti", "miercuri", "joi",
                      "vineri", "sambata", "duminica"};
```

Dacă examinați tipul matricei de la dreapta la stânga, veți observa că matricea conține șapte elemente. Asteriscul dinaintea numelui variabilei specifică un pointer. Dacă îl combinați cu numele de tip *char*, care precede numele variabilei, declararea va deveni o matrice *de pointeri la șiruri de caractere* (în exemplul precedent erau șapte șiruri). Una dintre matricele

de pointeri la șiruri de caractere cele mai utilizate este *argv*, care conține linia de comandă a programului dumneavoastră, așa cum va detalia capitolul acestei cărți despre linia de comandă.

Observație: *Atunci când declarați o matrice de pointeri la șiruri de caractere, compilatorul de C nu adaugă intrarea* ***NULL*** *pentru a indica sfârșitul de matrice, așa cum o face pentru șiruri de caractere.*

VIZUALIZAREA UNEI MATRICE DE ȘIRURI DE CARACTERE

C/C++ 522

Așa cum ați învățat, compilatorul de C tratează o matrice ca un pointer la locația de memorie de început a matricei. În secțiunile 521 ați creat o matrice șir de caractere numită *saptamana*, care conține zilele săptămânii. Atunci când creați o matrice de șiruri de caractere, compilatorul de C păstrează pointerii la șirurile matricei. Figura 522 ilustrează modul în care compilatorul de C păstrează matricea *litere* de mai jos:

```
char *litere[4] = {"AAA", "BBB", "CCC", "DDD"};
```

Observație: *Atunci când declarați o matrice de șiruri de caractere, compilatorul de C nu adaugă intrarea* ***NULL*** *pentru a indica sfârșitul de matrice, așa cum o face pentru șiruri de caractere.*

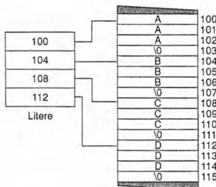


Figura 522 *Compilatorul de C păstrează matricele șir de caractere ca pe o matrice de pointeri.*

CICLAREA PRINTR-O MATRICE DE ȘIRURI DE CARACTERE

C/C++ 523

Așa cum ați învățat, atunci când creați o matrice de șiruri de caractere, compilatorul de C păstrează pointeri către fiecare șir din elementele matricei. Următorul program, *saptamana.c*, ciclează prin matricea *saptamana*, care conține pointeri la șirurile care conțin numele zilelor săptămânii, așa ca mai jos:

```
#include <stdio.h>

void main(void)
   {
     char *saptamana[7] = {"luni", "marti", "miercuri", "joi",
                          "vineri", "sambata", "duminica"};

     int i;
     for (i = 0; i < 7; i++)
       printf("saptamana[%d] contine %s\n", i, saptamana[i]);
   }
```

După cum vedeţi, programul ciclează pur şi simplu prin elementele matricei, utilizând specificatorul de format *%s* al funcţiei *printf*.

524 TRATAREA UNEI MATRICE ŞIR DE CARACTERE CA UN POINTER

C/C++

Aşa cum aţi învăţat, compilatorul de C tratează o matrice ca un pointer la elementul de început al matricei din memorie. Câteva dintre secţiunile prezentate în capitolul despre şiruri de caractere al acestei cărţi accesează matricele şiruri de caractere utilizând un pointer care este similar cu următorul:

```
char *sir;
```

Aţi învăţat, de asemenea, că compilatorul de C vă permite să creaţi matrice de şiruri de caractere. Următoarea declarare, de exemplu, creează o matrice numită *zilelucratoare*, care păstrează pointeri către cinci şiruri de caractere:

```
char *zilelucratoare[5];
```

Deoarece declararea creează o matrice, compilatorul de C vă permite să accesaţi matricea utilizând un pointer. Pentru a accesa matricea utilizând un pointer, trebuie să declaraţi o variabilă pointer, care indică la o matrice de şiruri de caractere. În cazul matricei *zilelucratoare*, declararea pointerului de referinţă ar deveni următoarea:

```
char **ptr_zilelucratoare;
```

Asteriscul dublu, în acest caz, arată că *ptr_zilelucratoare* este *un pointer la un pointer la un şir de caractere*. Câteva dintre secţiunile prezentate în capitolul despre linia de comandă din această carte lucrează cu un pointer la un pointer la un şir de caractere.

525 UTILIZAREA UNUI POINTER LA UN POINTER LA UN ŞIR DE CARACTERE

C/C++

În secţiunea 524, aţi învăţat că următoarea declarare creează un pointer la un pointer la un şir de caractere:

```
char **ptr_zilelucratoare;
```

Următorul program C, *zilelucr.c*, utilizează un pointer la un pointer la un șir de caractere pentru a afișa conținutul matricei *zilelucratoare*:

```
#include <stdio.h>

void main(void)
  {
    char *zilelucratoare[] = {"luni", "marti", "miercuri",
                              "joi", "vineri", ""};
    char **zile_lucru;

    zile_lucru = zilelucratoare;
    while (*zile_lucru)
      printf("%s\n", *zile_lucru++);
  }
```

La începutul rulării programului, el atribuie pointerului *zile_lucru* adresa de început a matricei *zilelucrătoare* (adresa șirului de caractere *luni*). Programul ciclează apoi până întâlnește pointerul la șirul *NULL* (condiția de sfârșit).

***Observație:** Atunci când declarați o matrice de pointeri la șiruri de caractere, compilatorul de C nu adaugă intrarea **NULL** pentru a indica sfârșitul de matrice, așa cum o face pentru șiruri de caractere. De aceea, declararea matricei **zilelucratoare** include în mod explicit un șir **NULL** astfel încât programul să poată testa bucla.*

DECLARAREA UNEI CONSTANTE ȘIR DE CARACTERE UTILIZÂND UN POINTER

C/C++526

Câteva dintre secțiunile prezentate de-a lungul acestei cărți au inițializat șiruri de caractere la declarare, cum arătăm în continuare:

```
char titlu[] = "Totul despre C/C++";
```

Atunci când declarați o matrice fără să treceți nimic între parantezele drepte, compilatorul de C alocă suficientă memorie pentru a putea păstra caracterele specificate (și terminatorul *NULL*), atribuind variabilei *titlu* un pointer la primul caracter. Deoarece compilatorul de C alocă automat memoria necesară și apoi lucrează cu pointerul către memorie, programele dumneavoastră pot utiliza un pointer șir de caractere, și nu o matrice, cum arătăm în continuare:

```
char *titlu = "Totul despre C/C++";
```

POINTERUL DE TIPUL VOID

C/C++527

Așa cum ați învățat, atunci când declarați o variabilă pointer, trebuie să specificați tipul valorii spre care indică pointerul (cum ar fi *int*, *float* sau *char*). Când faceți acest lucru, compilatorul poate să efectueze ulterior, în mod corect, operații aritmetice cu pointeri și să adauge corect valorile de deplasament atunci când incrementați sau decrementați pointerul. În unele cazuri, însă, programele dumneavoastră nu vor manipula oricum valoarea unui pointer. În schimb, programele vor intenționa să obțină numai un pointer către o locație de

memorie cu ajutorul căreia programul va da utilizarea pointerului. În astfel de cazuri, programele dumneavoastră pot crea un pointer de tip *void*, cum arătăm în continuare:

```
void *ptr_memorie;
```

Dacă veţi studia funcţiile bibliotecii run-time a limbajului C, prezentate în capitolul despre memorie al acestei cărţi, veţi găsi că unele dintre ele returnează pointeri de tip *void*. Astfel de funcţii vă indică, în esenţă, că returnează un pointer la o locaţie de memorie despre care compilatorul nu face nici un fel de presupuneri legate de conţinutul sau accesul la memorie.

528 CREAREA DE POINTERI LA FUNCŢII

C/C++

Aşa cum aţi învăţat, limbajul C vă permite să creaţi pointeri către orice tip de date (cum ar fi *int, float, char* şi chiar şir de caractere). În plus, limbajul C permite programelor dumneavoastră să creeze şi să utilizeze pointeri către funcţii. Cea mai obişnuită utilizare a pointerilor către funcţii este transmiterea unei funcţii ca parametru la o altă funcţie. Următoarele declarări creează pointeri către funcţii:

```
int (*min)();
int (*max)();
float (*medie)();
```

Observaţi utilizarea parantezelor în care se scrie numele variabilei. Dacă aţi înlătura parantezele, declarările ar servi drept prototipuri de funcţii pentru funcţii care returnează pointeri de un anumit tip, cum arătăm în continuare:

```
int *min();
int *max();
float *medie();
```

Atunci când citiţi declararea variabilei, începeţi cu declaraţia care apare între ultimele paranteze din interior şi continuaţi apoi de la dreapta la stânga:

```
int (*min)();
```

529 UTILIZAREA UNUI POINTER CĂTRE O FUNCŢIE

C/C++

În secţiunea 528 aţi învăţat că limbajul C vă permite să creaţi pointeri către funcţii. Cea mai obişnuită utilizare a pointerilor la funcţii este transmiterea unei funcţii ca un parametru la altă funcţie. Mai înainte, în acest capitol, aţi studiat funcţiile de sortare şi de căutare ale bibliotecii run-time de C. Aşa cum aţi învăţat, dacă vreţi să sortaţi valori de la cea mai mică la cea mai mare, veţi transmite o anumită funcţie la rutina bibliotecii run-time. Dacă vreţi să sortaţi valorile de la cea mai mare la cea mai mică, veţi transmite o altă funcţie. Următorul program, *trecfunc.c*, transmite fie funcţia *min*, fie funcţia *max* la funcţia *rezultat*. Valoarea returnată de funcţia *rezultat* va diferi în raport de funcţia transmisă de program:

```
#include <stdio.h>

int rezultat(int a, int b, int (*compar)())
  {
    return(compar(a, b)); // Invoca functia transmisa
  }

int max(int a, int b)
  {
    printf("In max\n");
    return((a > b) ? a: b);
  }

int min(int a, int b)
  {
    printf("In min\n");
    return((a < b) ? a: b);
  }

void main(void)
  {
    int rezult;
    rezult = rezultat(1, 2, &max);
    printf("Max lui 1 si 2 este %d\n", rezult);
    rezult = rezultat(1, 2, &min);
    printf("Min lui 1 si 2 este %d\n", rezult);
  }
```

UTILIZAREA UNUI POINTER LA UN POINTER LA UN POINTER

C/C++ 530

Așa cum ați învățat, limbajul C vă permite să creați variabile care sunt pointeri la alți pointeri. În general, nu există limite la numărul *redirectărilor* (pointer către pointer) pe care programele dumneavoastră pot să le utilizeze. Totuși, pentru majoritatea programatorilor, utilizarea mai mult decât *un pointer la un pointer* va crea o confuzie considerabilă și va face programul dumneavoastră foarte dificil de înțeles. De exemplu, următorul program, *ptrlaptr.c*, utilizează trei niveluri de pointeri la o valoare de tip *int*. Faceți-vă timp să urmăriți acest program și desenați nivelurile de redirectare pe hârtie până veți înțelege procesarea efectuată de programul *ptrlaptr.c*:

```
#include <stdio.h>

int care_e_valoarea(int ***ptr)
  {
    return(***ptr);
  }

void main(void)
  {
```

```
    int *nivel_1, **nivel_2, ***nivel_3, val = 1001;
    nivel_1 = &val;
    nivel_2 = &nivel_1;
    nivel_3 = &nivel_2;
    printf("Valoarea este %d\n", care_e_valoarea(nivel_3));
  }
```

531 STRUCTURILE

C/C++

Aşa cum aţi învăţat, o matrice este o variabilă care păstrează mai multe valori de acelaşi tip. Cu alte cuvinte, o matrice permite programelor dumneavoastră să grupeze informaţiile corelate într-o singură variabilă, cum ar fi 100 de rezultate la test sau 50 de salarii ale angajaţilor. Cum programele dumneavoastră devin mai complexe, există ocazii când veţi dori să grupaţi împreună informaţii corelate care diferă ca tip. De exemplu, presupunem că aveţi un program care lucrează cu informaţii despre angajaţi. Puteţi avea nevoie să ţineţi evidenţa următoarelor informaţii pentru fiecare angajat:

```
char nume[64];
int varsta;
char assn[11]; // numar asigurari sociale
int categ_salarizare;
float salariu;
unsigned nr_angajat;
```

Presupunem că aveţi câteva funcţii diferite în programul dumneavoastră care lucrează cu informaţiile despre angajaţi. De fiecare dată când programul dumneavoastră invocă funcţia, trebuie să vă asiguraţi că aţi specificat toţi parametrii, în ordinea corectă. Aşa cum am discutat în capitolul despre funcţii al acestei cărţi, cu cât programul dumneavoastră transmite mai mulţi parametri la funcţii, cu atât va fi mai greu de înţeles şi va creşte posibilitatea de apariţie a erorilor. Pentru a reduce complexitatea, programele dumneavoastră pot crea o *structură* care grupează informaţiile corelate într-o singură variabilă. De exemplu, următoarea declarare de structură creează o structură, numită *Angajat*, care conţine câmpurile amintite mai sus:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
  };
```

Aşa cum veţi învăţa în următoarele secţiuni, această declarare creează structuri de tip *Angajat*.

O STRUCTURĂ ESTE UN ȘABLON PENTRU DECLARĂRILE DE VARIABILE

C/C++ 532

În secțiunea 531 ați învățat că limbajul C vă permite să grupați informațiile corelate într-o structură. Prin ea însăși, definirea structurii nu creează nici o variabilă. În schimb, definirea specifică un șablon pe care programul dumneavoastră poate să îl folosească mai târziu pentru a declara variabile. De aceea, definirea unei structuri nu alocă memorie. În schimb, compilatorul, pur și simplu, ia notă de existența definirii, în cazul în care programul declară mai târziu o variabilă de tipul structurii.

NUMELE GENERIC AL UNEI STRUCTURI ESTE NUMELE STRUCTURII

C/C++ 533

În secțiunea 531 ați învățat că limbajul C vă permite să grupați variabilele corelate în structuri. Utilizând cuvântul cheie *struct*, programele dumneavoastră pot declara o structură, cum arătăm în continuare:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
  };
```

În exemplul precedent, numele structurii este *Angajat*. Programatorii de C se referă la numele structurii ca la un *nume generic*. Așa cum veți învăța în secțiunea 534, programele dumneavoastră pot utiliza numele generic al structurii pentru a declara variabile de un anumit tip. Următoarea declarație creează o structură numită *Schita*:

```
struct Schita
  {
    int tip; // 0 = cerc, 1 = patrat, 2 = triunghi
    int culoare;
    float raza;
    float aria;
    float perimetru;
  };
```

DECLARAREA UNEI VARIABILE STRUCTURĂ ÎN ALT MOD

C/C++ 534

În secțiunea 531 ați învățat că limbajul C vă permite să grupați informațiile corelate într-o structură. Așa cum ați învățat, definirea unei structuri nu creează prin ea însăși o variabilă utilizabilă. Mai curând, definirea servește pur și simplu ca șablon pentru viitoare declarări de variabile. Limbajul C oferă două căi de declarare a variabilelor de un anumit tip structură.

Prima, presupune ca programul dumneavoastră să declare o structură de tip *Angajat*, cum arătăm în continuare:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
  };
```

După definirea structurii, programele dumneavoastră pot declara variabile de tip *Angajat*, ca mai jos:

```
struct Angajat angajati_info;
struct Angajat noi_angajati, fosti_angajati;
```

Prin cealaltă cale, limbajul C vă permite să declarați variabile de tip structură, urmând definirea structurii, cum arătăm în continuare:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
  } angajati_info, noi_angajati, fosti_angajati;
```

535 SĂ ÎNȚELEGEM MEMBRII STRUCTURII



Așa cum ați învățat, limbajul C vă permite să grupați informații corelate în cadrul structurilor. De exemplu, următoarea instrucțiune creează variabila numită *triunghi* utilizând structura *Schita*:

```
struct Schita
  {
    int tip; // 0 = cerc, 1 = patrat, 2 = triunghi
    int culoare;
    float raza;
    float aria;
    float perimetru;
  } triunghi;
```

Fiecare parte de informație din structură este un *membru*. În cazul structurii *Schita*, există cinci membri: *tip, culoare, raza, aria* și *perimetru*. Pentru a accesa un anumit membru,

utilizați operatorul C *punct* (.). De exemplu, următoarea instrucțiune atribuie valori diferiților membri ai variabilei *triunghi*:

```
triunghi.tip = 2;
triunghi.perimetru = 30.0;
triunghi.aria = 45.0;
```

VIZUALIZAREA UNEI STRUCTURI

C/C++ 536

Așa cum ați învățat, limbajul C vă permite să grupați informațiile corelate în structuri. Atunci când declarați o variabilă de un anumit tip de structură, compilatorul de C alocă suficientă memorie pentru a păstra valorile pentru fiecare membru al structurii. De exemplu, dacă declarați o structură de tip *Angajat*, compilatorul de C va aloca memoria cum este arătat în figura 536.

```
struct Angajat{
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
};
```

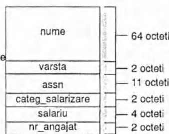


***Figura 536** O hartă logică a memoriei alocate de compilatorul de C pentru păstrarea structurii.*

UTILIZAREA STRUCTURII

C/C++ 537

Așa cum ați învățat, C vă permite să grupați informațiile corelate într-o structură. În capitolul despre dată și oră al acestei cărți, veți utiliza funcția *getdate* pentru a determina data curentă a sistemului. Funcția atribuie data curentă membrilor unei structuri de tip *date*, cum arătăm în continuare:

```
struct date
  {
    int da_year;      // anul curent
    int da_day;       // ziua din luna
    int da_mon;       // luna anului
  };
```

Următorul program, *datados.c*, utilizează funcția *getdate* pentru a atribui data variabilei *data_curenta*:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
```

```
    struct date data_curenta;
    getdate(&data_curenta);
    printf("Data curenta: %d-%d-%d\n", data_curenta.da_mon,
           data_curenta.da_day, data_curenta.da_year);
  }
```

Deoarece funcţia trebuie să modifice valoarea parametrului, programul transmite variabila structură către funcţie prin referinţă (prin adresă).

538 TRANSMITEREA UNEI STRUCTURI CĂTRE O FUNCŢIE



Aşa cum aţi învăţat, limbajul C vă permite să grupaţi informaţiile corelate într-o structură. La fel ca orice variabilă, compilatorul de C permite transmiterea unei structuri către o funcţie. Următorul program, *strufunc.c*, transmite o structură de tip *Schita* către funcţia *o_structura* care la rândul său, afişează fiecare membru al structurii:

```
#include <stdio.h>
struct Schita
  {
    int tip;
    int culoare;
    float raza;
    float aria;
    float perimetru;
  };
void o_structura(struct Schita schita)
  {
    printf("schita.tip %d\n", schita.tip);
    printf("schita.culoare %d\n", schita.culoare);
    printf("schita.raza %f schita.aria %f schita.perimetru
      %f\n", schita.raza, schita.aria, schita.perimetru);
  }
void main(void)
  {
    struct Schita cerc;
    cerc.tip = 0;
    cerc.culoare = 1;
    cerc.raza = 5.0;
    cerc.aria = 22.0 / 7.0 * cerc.raza * cerc.raza;
    cerc.perimetru = 2.0 * 22.0 / 7.0 * cerc.raza;
    o_structura(cerc);
  }
```

MODIFICAREA UNEI STRUCTURI ÎN CADRUL UNEI FUNCȚII

C/C++539

În secțiunea 538 ați învățat că puteți transmite structuri către funcții așa cum puteți transmit variabilele de orice tip. Pentru a modifica membrii structurii în cadrul unei funcții, trebuie s transmiteți structura prin adresă (la fel cum transmiteți o variabilă a cărei valoare vreți să modificați). Următorul program, *schstruc.c*, invocă funcția *modific_structura* care modific valorile conținute într-o structură de tip *Schita*:

```
#include <stdio.h>
struct Schita
  {
    int tip;
    int culoare;
    float raza;
    float aria;
    float perimetru;
  };
void modific_structura(struct Schita *schita)
  {
    (*schita).tip = 0;
    (*schita).culoare = 1;
    (*schita).raza = 5.0;
    (*schita).aria = 22.0 / 7.0 * (*schita).raza * (*schita).raza;
    (*schita).perimetru = 2.0 * 22.0 / 7.0 * (*schita).raza;
  }
void main(void)
  {
    struct Schita cerc;
    modific_structura (&cerc);
    printf("cerc.tip %d\n", cerc.tip);
    printf("cerc.culoare %d\n", cerc.culoare);
    printf("cerc.raza %f cerc.aria %f cerc.perimetru %f\n",
           cerc.raza, cerc.aria, cerc.perimetru);
  }
```

Pentru a modifica membrii structurii, programul transmite către funcție un pointer l structură. În cadrul funcției, instrucțiunile dereferențiază membrii pointerilor utilizân operatorul *de redirectare asterisc*:

```
(*pointer).membru = valoare;
```

540 REDIRECTAREA (*POINTER).MEMBRU



Pentru a modifica un membru al unei structuri în cadrul unei funcții, programul trebuie să transmită un pointer la structură. În cadrul funcției, instrucțiunile dereferențiază pointerul utilizând operatorul *de redirectare asterisc*, cum arătăm în continuare:

```
(*pointer).membru = valoare;
```

Pentru a rezolva pointerul, compilatorul de C începe cu parantezele, obținând mai întâi locația structurii. Apoi, el adaugă la adresă deplasamentul membrului specificat. Dacă omiteți parantezele, compilatorul de C presupune că însuși membrul este un pointer și utilizează operatorul de redirectare asterisc pentru a-l rezolva, cum arătăm în continuare:

```
*pointer.membru = valoare;
```

Sintaxa cu omisiunea parantezelor ar fi corectă pentru o structură care are un membru pointer, cum ar fi următoarea:

```
struct Planeta
  {
    char nume[48];
    int *un_pointer;
  } planeta;
```

După cum vedeți, al doilea membru este un pointer la o valoare de tip *int*. Presupunând că programul a atribuit anterior pointerul la o locație de memorie, următoarea instrucțiune plasează valoarea 5 în locația de memorie:

```
*planeta.un_pointer = 5;
```

541 FORMATUL POINTER->MEMBRU



În secțiunea 540 ați învățat că pentru a modifica un membru al unei structuri în cadrul unei funcții, programul trebuie să transmită un pointer la structură. Pentru a dereferenția pointerul în cadrul funcției, limbajul C dispune de două formate. Primul, așa cum ați văzut, este referința la membrul structurii în următorul mod:

```
(*pointer).membru = valoare;
o_valoare = (*pointer).membru;
```

A doua, este de următorul format:

```
pointer->membru = valoare;
o_valoare = pointer->membru;
```

Următorul program, *schmemb.c*, utilizează al doilea format în cadrul funcției *modific_structura* pentru a referenția membrii unei structuri transmisă către funcție prin adresă:

```
#include <stdio.h>
struct Schita
```

```
   {
    int tip;
    int culoare;
    float raza;
    float aria;
    float perimetru;
   };

void modific_structura(struct Schita *schita)
   {
    schita->tip = 0;
    schita->culoare = 1;
    schita->raza = 5.0;
    schita->aria = 22.0 / 7.0 * schita->raza * schita->raza;
    schita->perimetru = 2.0 * 22.0 / 7.0 * schita->raza;
   }

void main(void)
   {
    struct Schita cerc;

    modific_structura (&cerc);
    printf("cerc.tip %d\n", cerc.tip);
    printf("cerc.culoare %d\n", cerc.culoare);
    printf("cerc.raza %f cerc.aria %f cerc.perimetru %f\n",
           cerc.raza, cerc.aria, cerc.perimetru);
   }
```

UTILIZAREA UNEI STRUCTURI FĂRĂ NUME GENERIC

C/C++ 542

Aşa cum aţi învăţat, numele generic al unei structuri este numele structurii. Utilizând numele generic, programele dumneavoastră pot declara variabile de un anume tip de structură. Totuşi, atunci când declaraţi variabile de tip structură care urmează imediat după definirea structurii, nu este necesar să specificaţi un nume generic. De exemplu, următoarea declaraţie creează două variabile structuri:

```
struct
   {
    int tip; // 0 = cerc, 1 = patrat, 2 = triunghi
    int culoare;
    float raza;
    float aria;
    float perimetru;
   } triunghi, cerc;
```

Dacă programul dumneavoastră nu va face mai târziu referinţă la structură prin nume (ca într-un prototip de funcţie sau în parametri formali), atunci puteţi să omiteţi numele generic

al structurii, cum am arătat. Dacă, însă, includeți numele generic, dați altor programatori care vă citesc programul posibilitatea înțelegerii scopului structurii. Atunci când dați un nume generic semnificativ, faceți ca programele dumneavoastră să fie mai lizibile.

543 DOMENIUL DEFINIRII UNEI STRUCTURI



În capitolul despre funcții al acestei cărți, ați învățat că *domeniul de valabilitate* definește regiunea programului în care un identificator (cum ar fi o variabilă sau o funcție) este recunoscută. Atunci când definiți o structură, trebuie să țineți seama de domeniul de valabilitate al structurii. Dacă analizați programul anterior care lucrează cu structuri în cadrul funcțiilor, veți observa că programul definește structura în afara și înaintea funcției care o utilizează. Ca urmare, definițiile structurii au un domeniu de *valabilitate global*, care permite tuturor funcțiilor care urmează să facă referire la ele. Dacă, în schimb, programul ar fi definit structura în cadrul funcției *main*, unica funcție care ar cunoaște existența structurii ar fi funcția *main*. Dacă aveți nevoie ca mai multe funcții ale programului dumneavoastră să utilizeze o definire de structură, trebuie să definiți structura în afara funcțiilor dumneavoastră ceva mai înainte de toate funcțiile care trebuie să acceseze structura.

544 INIȚIALIZAREA UNEI STRUCTURI



Așa cum ați învățat, limbajul C vă permite să inițializați matricele când le declarați. În același mod, programele dumneavoastră pot, de asemenea, să inițializeze o structură la declararea ei. Următorul program, *initstru.c*, declară și inițializează o structură de tip *Schita*:

```
#include <stdio.h>

void main(void)
  {
    struct Schita
  {
        int tip;
        int culoare;
        float raza;
        float aria;
        float perimetru;
      }; cerc = {0, 1, 5.0, 78.37, 31.42};

    printf("cerc.tip %d\n", cerc.tip);
    printf("cerc.culoare %d\n", cerc.culoare);
    printf("cerc.raza %f cerc.aria %f cerc.perimetru %f\n",
            cerc.raza, cerc.aria, cerc.perimetru);
  }
```

Deoarece programul utilizează structura numai în funcția *main*, programul definește structura în cadrul funcției *main*.

EFECTUAREA DE OPERAȚII I/O CU STRUCTURI

C/C++ 545

Câteva dintre secțiunile prezentate în acest capitol au utilizat funcția *printf* pentru a afișa valoarea unuia sau a mai multor membri ai unei structuri. Atunci când efectuați operații de I/O la ecran sau tastatură care afectează membrii structurii, trebuie să efectuați operațiile de I/O membru cu membru. Atunci însă când citiți sau scrieți structuri dintr-un sau într-un fișier, programele dumneavoastră pot lucra cu întreaga structură. Dacă programul dumneavoastră utilizează fluxurile de fișiere, puteți utiliza funcțiile *fwrite* și *fread* pentru a citi și scrie structuri. Capitolul despre fișiere al acestei cărți ilustrează modul de utilizare a funcțiilor *fwrite* și *fread* pentru a efectua operații de I/O cu structuri. Pentru a înțelege mai bine acest proces, apelați la programele *dtoutf.c și dtinf.c* din directorul Tip0545 al compact discului care însoțește cartea. Dacă programul dumneavoastră utilizează indicatoare de fișier, puteți utiliza funcțiile *read* și *write* pentru a efectua operații de I/O cu structuri. Fișierele *dtout.c* și *dtin.c* (din directorul Tip 0545) aflate în compact discul însoțitor, ilustrează cum pot utiliza programele dumneavoastră funcțiile *read* și *write* pentru a efectua I/O cu structuri. Fiecare dintre funcțiile de I/O pe care tocmai le-am prezentat citesc sau scriu un interval de octeți. Când compilatorul de C stochează o structură în memorie, structura este de fapt chiar un interval de octeți. De aceea, pentru a utiliza o structură cu aceste funcții, transmiteți pur și simplu un pointer la structură, cum arătăm în exemplele de programe.

UTILIZAREA UNEI STRUCTURI IMBRICATE

C/C++ 546

Așa cum ați învățat, compilatorul de C vă permite să stocați informații corelate în cadrul unei structuri. Într-o structură, puteți include membri de orice tip (*int*, *float* și așa mai departe), precum și membri care sunt ei înșiși structuri. De exemplu, următoarea declarare de structură include o structură de tip *Date* care conține date despre angajări:

```
struct Angajat
  {
    char nume[64];
    int varsta;
    char assn[11]; // numar asigurari sociale
    struct Date
      {
        int day;
        int mon;
        int year;
      } date_angaj;
    int categ_salarizare;
    float salariu;
    unsigned nr_angajat;
  } noi_angajati;
```

Pentru a accesa membrii unei structuri imbricate, utilizați operatorul *punct*, mai întâi pentru a specifica structura imbricată și apoi pentru a specifica membrul dorit, cum arătăm în continuare:

```
noi_angajati.date_angaj.month = 12;
```

547 STRUCTURI CARE CONȚIN MATRICE



Aşa cum aţi învăţat, membrii structurii pot fi de orice tip, inclusiv structuri sau matrice. Atunci când un membru al structurii este o matrice, programele dumneavoastră referenţiază membrul matrice ca şi cum ar fi o matrice oarecare, cu excepţia că numele variabilei şi operatorul *punct* vor precede numele matricei. De exemplu, următorul program, *structab.c*, iniţializează câteva câmpuri de structură, inclusiv o matrice. Programul ciclează apoi prin elementele matricei, afişând valoarea lor:

```
#include <stdio.h>

void main(void)
   {
     struct Date
       {
         char nume_luna[64];
         int month;
         int day;
         int year;
       } data_curenta = { "iulie", 7, 4, 1994 };
     int i;

     for (i = 0; data_curenta.nume_luna[i]; i++)
       putchar(data_curenta.nume_luna[i]);
   }
```

548 CREAREA UNEI MATRICE DE STRUCTURI



Aşa cum aţi învăţat, o matrice permite programelor dumneavoastră să păstreze mai multe valori de acelaşi tip. Majoritatea matricelor prezentate de-a lungul acestei secţiuni au fost de tip *int, float* sau *char*. Limbajul C vă permite, însă, să declaraţi matrice de un tip specificat de structură. De exemplu, următoarea declarare creează o matrice aptă să păstreze informaţii despre 100 de angajaţi:

```
struct Angajat
   {
     char nume[64];
     int varsta;
     char assn[11];   // numar asigurari sociale
     int categ_salarizare;
     float salariu;
     unsigned nr_angajat;
   } personal[100];
```

Presupunând că programul a atribuit valori pentru fiecare angajat, următoarea buclă *for* va afişa numele şi numărul fiecărui angajat:

```
for (angaj = 0; angaj < 100; angaj++)
   printf("Angajat: %s Numar: %d\n", personal[angaj].nume,
     personal[angaj].nr_angajat);
```

Atunci când utilizaţi o matrice de structuri, pur şi simplu adăugaţi operatorul *punct* la fiecare element al matricei.

SERVICIILE SISTEMULUI DOS

C/C++ 549

După cum ştiţi, DOS este un sistem de operare pentru calculatoarele compatibile cu IBM PC. Sistemul DOS vă permite rularea programelor şi păstrează informaţia pe disc. În plus, sistemul DOS vă pune la dispoziţie serviciile care permit programelor să aloce memorie, să acceseze dispozitive, cum ar fi imprimanta, şi să gestioneze alte resurse ale sistemului. Pentru a ajuta programele dumneavoastră să beneficieze de avantajele caracteristicilor conţinute în sistemul DOS – cum ar fi determinarea volumului de spaţiu liber de pe disc, crearea sau selectarea unui director sau chiar captarea intrărilor de la tastatură – sistemul DOS dispune de un set de servicii pe care programele dumneavoastră pot să le utilizeze. Spre deosebire de funcţiile puse la dispoziţie de biblioteca run-time a limbajului C, programele dumneavoastră nu accesează serviciile DOS utilizând o simplă interfaţă de apelare a funcţiei. În schimb, programatorii scriu serviciile astfel încât alţi programatori să poată accesa serviciile la nivelul limbajului de asamblare, utilizând registre şi întreruperi. Însă, aşa cum veţi învăţa în acest capitol, limbajul C facilitează de fapt accesul programelor la avantajele serviciilor DOS, fără să vă oblige să utilizaţi limbajul de asamblare. În plus, biblioteca *run-time* a limbajului C oferă adesea o interfaţă la multe servicii DOS, prin intermediul funcţiilor.

Atunci când creaţi programele dumneavoastră proprii, uneori veţi avea de ales între utilizarea unei funcţii din biblioteca *run-time* de C şi un serviciu DOS. De regulă, este bine să utilizaţi funcţiile bibliotecii *run-time* ori de câte ori este posibil, în locul serviciilor DOS, pentru că utilizând funcţiile bibliotecii *run-time* va creşte portabilitatea programelor dumneavoastră. În anumite cazuri, funcţiile bibliotecii *run-time* de C puse la dispoziţie de mediul UNIX sau Windows pot face ceea ce compilatorul DOS oferă. Atunci când utilizaţi o funcţie a bibliotecii *run-time* de C – şi nu un serviciu DOS – nu aveţi nevoie să modificaţi programul pentru a-l rula sub UNIX. În schimb, trebuie numai să îl recompilaţi.

Atunci când programaţi pentru Windows, veţi utiliza diferite seturi de servicii denumite servicii ale sistemului Windows (cunoscute, de asemenea, ca Interfaţa API – Application Programming Interface). Interfaţa API din Windows vă permite să controlaţi majoritatea serviciilor din Windows, inclusiv serviciile de fişier, serviciile de memorie şi aşa mai departe, utilizând setul de funcţii al limbajului C++. Programele dumneavoastră pot invoca funcţiile C++ din cadrul codului lor obişnuit şi returna valori în mod asemănător cu serviciile DOS. Veţi învăţa mai multe despre serviciile de sistem Windows în secţiunile 1251-1500.

SERVICIILE BIOS

C/C++ 550

BIOS desemnează *serviciile de intrare-ieşire de bază* (*Basic Input-Output Services*). Pe scurt, BIOS este un cip în cadrul calculatorului dumneavoastră care conţine instrucţiunile pe care calculatorul le utilizează pentru a scrie pe ecran sau la imprimantă, pentru a citi caractere de la tastatură sau pentru a citi şi scrie pe disc. Ca şi în cazul serviciilor DOS, programele

dumneavoastră pot utiliza serviciile BIOS pentru a efectua diferite operații. De exemplu, puteți utiliza serviciile BIOS pentru a determina numărul de porturi paralele și seriale, tipul monitorului sau numărul de unități de disc disponibile. La fel ca în cazul serviciilor DOS, programatorii au proiectat rutinele BIOS pentru a fi utilizate de programe în limbaj de asamblare. Majoritatea compilatoarelor de C dispun, totuși, de funcții ale bibliotecii *run-time* care permit programelor dumneavoastră să utilizeze aceste servicii fără a avea nevoie de limbajul de asamblare. Majoritatea programatorilor confundă serviciile DOS cu serviciile BIOS. Așa cum arată figura 550.1, BIOS este situat imediat deasupra componentei hardware a calculatorului dumneavoastră. Serviciile DOS sunt situate deasupra serviciilor BIOS, iar programele dumneavoastră, deasupra sistemului DOS.

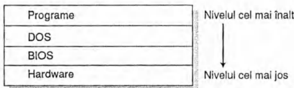


Figura 550.1 *Relațiile dintre BIOS, DOS și programe.*

Uneori, însă, DOS și chiar programele dumneavoastră pot evita serviciile BIOS și pot accesa direct componenta hardware. O aplicație care trebuie să dispună rapid de imagini video, de exemplu, poate evita serviciile DOS și BIOS pentru a lucra direct cu memoria video. De regulă, însă, numai programatorii experimentați ar trebui să evite serviciile DOS și BIOS. Sistemele DOS și BIOS execută numeroase testări de erori, ceea ce simplifică sarcinile programatorului.

Toate versiunile de Windows, inclusiv Windows 95 și Windows NT, vor apela propriile lor servicii de sistem. Însă, întocmai ca în cazul serviciilor de sistem DOS, serviciile de sistem Windows apelează până la urmă serviciile de nivel BIOS pentru a accesa componenta hardware a calculatorului. Totuși, în general, nu este o idee bună ca atât timp cât este posibil să se evite serviciile de sistem Windows să se apeleze direct serviciile de nivel BIOS, datorită modului în care este creat sistemul de operare Windows. În mod obișnuit, veți obține cele mai bune rezultate în cadrul programelor Windows utilizând interfața API Windows și ar trebui să apelați serviciile BIOS numai dacă este absolut necesar. Figura 550.2 arată relațiile dintre BIOS, DOS, Windows și programe.

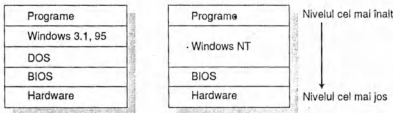


Figura 550.2 *Relațiile dintre BIOS, DOS, Windows și programe.*

REGISTRELE

C/C++551

Atunci când programul dumneavoastră se execută, el trebuie să fie localizat în memoria calculatorului. Unitatea centrală de procesare (CPU) a calculatorului dumneavoastră va da instrucțiuni și date ale programului dumneavoastră atunci când acesta are nevoie, din memorie. Pentru a îmbunătăți performanța, CPU conține câteva locații temporare de păstrare numite *registre*. Deoarece registrele sunt plasate chiar în CPU, CPU poate accesa foarte repede conținutul fiecărui registru. În general, CPU utilizează patru tipuri de registre *segment, deplasament, de uz general, indicatori*. Tabelul 551 descrie pe scurt utilizarea fiecărui tip de registru.

Tip registru	Utilizare
Segment	Păstrează memoria adresei de început a unui bloc de memorie, cum ar fi începutul codului programului dumneavoastră sau ale datelor lui.
Deplasament	Păstrează deplasamentul de 16 octeți într-un bloc de memorie, cum ar fi locația unei anumite variabile în cadrul segmentului de date al programului dumneavoastră.
De uz general	Păstrează temporar datele programului.
Indicator	Conține starea procesorului și informațiile despre erori.

Tabelul 551 *Tipurile de registre ale PC-ului.*

PC-ul utilizează o valoare *segment* și *deplasament* pentru a localiza articolele în memorie Atunci când utilizați serviciile DOS, puteți să atribuiți adresele de *segment* și *deplasament* ale uneia sau mai multor variabile la diferite registre *segment*. PC-ul dispune de patru registre *de uz general*, numite *AX, BX, CX* și *DX*. Fiecare registru *general* poate să păstreze 16 biți de date (2 octeți). În anumite cazuri, puteți să păstrați un singur octet de informație în interiorul registrului. Pentru a vă ajuta să faceți aceasta, PC-ul vă permite să accesați fiecare dintre octeții inferior și superior ai registrului utilizând numele arătate în figura 551.1.

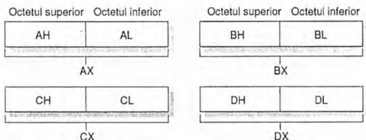


Figura 551.1 *Cei patru registre de uz general ai PC-ului.*

Atunci când utilizați serviciile DOS și BIOS, veți plasa parametrii pentru oricare serviciu pe care programul dumneavoastră îl apelează în cadrul registrelor *de uz general*. Când serviciul se încheie, DOS și BIOS pot plasa rezultatul serviciului într-unul dintre registrele de uz general. În sfârșit, registrului indicator vor păstra starea pentru CPU și posibilele valori ale stării de eroare. Când serviciile DOS și BIOS se încheie, deseori ele dau valoarea 1 sau 0 diferiților biți din cadrul registrului indicator pentru a indica succesul sau eroarea. Figura 551.2 ilustrează biții din cadrul registrului indicator (căsuțele gri reprezintă biți neutilizați).

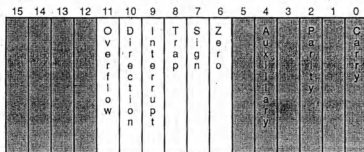


Figura 551.2 Biții în cadrul registrului indicator al PC-ului.

Câteva dintre secțiunile prezentate pe parcursul acestui capitol abordează mai detaliat registrele *segment, deplasament* în detaliu.

552 REGISTRUL INDICATOR

C/C++

Așa cum ați învățat, registrul indicator conține starea pentru CPU și informații despre erori. După ce PC-ul încheie diferite operații, cum ar fi adunarea, scăderea, compararea, el fixează diferiți biți în registrul indicator. De asemenea, multe dintre serviciile DOS și BIOS dau valoarea 1 indicatorului carry pentru a indica o eroare. Tabelul 552 descrie biții pe care serviciile DOS și BIOS îi utilizează în cadrul registrului indicator.

Bit	Indicator	Semnificație
0	Carry	Indică un transport aritmetic
2	Parity	Indică o operație aritmetică ce are ca rezultat un număr par de biți 1
4	Auxiliary	Indică o ajustare necesară ca urmare a unei operații aritmetice în BCD (*binary coded decimal* = binar codificat zecimal)
6	Zero	Indică rezultatul 0 al unei comparații sau operații aritmetice
7	Sign	Indică un rezultat negativ
8	Trap	Utilizat pentru identificarea erorilor prin depanare
10	Direction	Controlează direcția instrucțiunilor pentru șir de caractere
11	Overfow	Indică o depășire aritmetică

Tabelul 552 Biți utilizați de registrul indicator.

Atunci când utilizați serviciile DOS și BIOS în cadrul programului dumneavoastră, asigurați-vă că programul testează bitul indicator pe care serviciul îl stabilește pentru a determina dacă serviciul a fost îndeplinit cu succes sau nu.

553 ÎNTRERUPERILE SOFTWARE

C/C++

O *întrerupere* apare când CPU trebuie temporar să oprească ceea ce execută, astfel încât să poată executa o altă operație. Când operația ia sfârșit, CPU reia lucrul inițial ca și cum nu ar fi fost oprit. Există două feluri de întreruperi: întreruperi hardware și întreruperi software.

Dispozitivele conectate la sau în interiorul calculatorului, cum ar fi ceasul, unitatea de disc sau tastatura provoacă întreruperi. Creatorii PC-ului l-au prevăzut să accepte 256 de întreruperi, numerotate de la 0 la 255. Deoarece componenta hardware a calculatorului are nevoie numai de un număr mic din aceste întreruperi, multe dintre ele sunt disponibile pentru utilizarea prin software. Serviciile BIOS, de exemplu, utilizează întreruperi de la 5 şi 10H până la 1FH (16 în zecimal până la 31 în zecimal). De asemenea, serviciile DOS folosesc întreruperi de la 21H la 28H (33 în zecimal până la 40 în zecimal) şi 2FH (47 în zecimal).

Când scrieţi programele în limbaj de asamblare, atribuiţi parametri regiştrelor PC-ului şi apoi invocaţi întreruperea care corespunde serviciului de sistem pe care îl doriţi. De exemplu, BIOS utilizează întreruperea 10H pentru a accesa imagini video. Pentru a afişa o literă pe ecran, de exemplu, atribuiţi litera pe care o vreţi registrului AL, atribuiţi valoarea 9H registrului AH (care indică BIOS-ului să efectueze operaţia de scriere video), atributul pe care îl doriţi (îngroşat, cu clipire intermitentă, normal şi aşa mai departe) – registrului BX şi apoi invocaţi întreruperea INT 10H, cum arătăm în continuare:

```
MOV AL, 41 ; A este caracterul ASCII 41H
MOV AH,9 ; Solicitare scriere video
MOV BX,7 ; Atributul caracterului
MOV CX,1 ; Numarul de caractere de scris
INT 10 ; Executa serviciul video
```

Aşa cum aţi învăţat, majoritatea serviciilor DOS utilizează întreruperea INT 21H. Din fericire, nu trebuie să lucraţi în limbaj de asamblare în cadrul programelor în C pentru a invoca un serviciu.

Observaţie: *Deşi puteţi utiliza întreruperile software în cadrul programelor Windows, majoritatea activităţilor din aceste programe trebuie să utilizeze interfaţa Windows API.*

UTILIZAREA SERVICIILOR BIOS PENTRU ACCESUL LA IMPRIMANTĂ

C/C++ 554

Câteva dintre secţiunile prezentate pe parcursul acestei cărţi au scris ieşirea la imprimantă utilizând indicatorul de fişierul *stdprn*. Înainte ca programele dumneavoastră să execute operaţiile de I/O la imprimantă, puteţi totuşi să verificaţi dacă imprimanta este conectată şi dacă are hârtie. Pentru a face aceasta, programele dumneavoastră pot utiliza funcţia *biosprint*. Veţi implementa funcţia *biosprint*, aşa cum arătăm în continuare:

```
#include <bios.h>
int biosprint(int comanda, int octet, int nr_port);
```

Parametrul *comanda* specifică una dintre operaţiile listate în tabelul 554.1.

Comanda	Semnificaţie
0	Tipăreşte octetul specificat
1	Iniţializează portul imprimantei
2	Citeşte starea imprimantei

Tabelul 554.1 *Valorile posibile pentru parametrul **comanda**.*

Dacă tipăriți un caracter, parametrul *octet* specifică valoarea ASCII sau ASCII extins sau caracterul pe care îl doriți. Parametrul *nr_port* specifică portul imprimantei pe care doriți să realizați tipărirea, care poate fi 0 pentru LPT1, 1 pentru LPT2 și așa mai departe.

Funcția *biosprint* returnează o valoare întreagă în intervalul de la 0 la 255, ai cărei biți sunt definiți în tabelul 554.2.

Bitul	Semnificație
0	Dispozitiv în pauză
3	Eroare de I/O
4	Imprimantă selectată
5	Lipsă hârtie
6	Confirmare dispozitiv
7	Dispozitivul nu este ocupat

Tabelul 554.2 *Biții returnați de funcția* ***biosprint***.

Următorul program, *printtst.c*, utilizează funcția *biosprint* pentru a testa repetat starea imprimantei, până când apăsați o tastă oarecare. Rulați următorul program și experimentați-l cu imprimanta dumneavoastră, deconectând-o, scoțând hârtia și așa mai departe. Când faceți acestea, programul trebuie să afișeze diferite mesaje pe ecran, cum arătăm în continuare:

```
#include <bios.h>
#include <conio.h>
#include <stdio.h>

void main(void)
  {
    int stare = 0;
    int stare_veche = 0;

    do
     {
       stare = biosprint(2, 0, 0); // Citeste LPT1
       if (stare != stare_veche)
         {
           if (stare & 1)
            printf ("Pauza\t");
           if (stare & 8)
            printf ("Eroare de iesire\t");
           if (stare & 16)
            printf ("Imprimanta selectata\t");
           if (stare & 32)
            printf ("Lipsa hartie\t");
           if (stare & 64)
            printf ("Confirmat\t");
           if (stare & 128)
            printf ("Imprimanta nu e ocupata");
           printf ("\n");
           stare_veche = stare;
```

```
      }
    }
  while (! kbhit());
}
```

***Observație:** Multe compilatoare dispun de funcția numită **_bios_printer** care este similară cu **biosprint**. Pentru a accesa imprimanta conectată la portul serial, trebuie să utilizați funcția **_bios_serialcom**.*

***Observație:** Compact discul însoțitor al acestei cărți include programul **win_print.cpp** care utilizează interfața Windows API pentru a trimite informații către imprimantă.*

INFORMAȚIA CTRL+BREAK

C/C++ 555

Când lucrați în mediu DOS, comanda DOS BREAK vă permite să activați sau să dezactivați testarea extinsă pentru CTRL+BREAK. Atunci când activați testarea extinsă, DOS mărește numărul de operații după care testează dacă utilizatorul a apăsat combinația de taste CTRL+C sau CTRL+BREAK. Atunci când dezactivați testarea extinsă pentru CTRL+BREAK, DOS testează numai un CTRL+BREAK după efectuarea unei intrări/ieșiri de la tastatură, ecran sau imprimantă. Multe compilatoare de C dispun de două funcții, *getcbrk* și *setcbrk*, pe care programele dumneavoastră le pot utiliza pentru a obține și a stabili starea testării pentru CTRL+BREAK. Veți implementa funcțiile *getcbrk* și *setcbrk*, ca mai jos:

```
#include <dos.h>

int getcbrk(void);
int setcbrk(int set);
```

Funcția *getcbrk* returnează valoarea 0 dacă ați dezactivat testarea extinsă pentru CTRL+BREAK și 1 dacă ea este activă. La fel, funcția *setcbrk* utilizează valorile 0 și 1, respectiv, pentru a dezactiva și a activa testarea extinsă. Funcția *setcbrk* returnează, de asemenea, valoarea 0 sau 1, în funcție de starea testării extinse pe care ați selectat-o. Următorul program, *ctrlbrk.c*, utilizează funcția *setcbrk* pentru a dezactiva testarea pentru CTRL+BREAK. Programul folosește valoarea returnată a funcției *getcbrk* pentru a afișa valoarea anterioară, ca mai jos:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    printf("Starea extinsa anterioara pentru Ctrl-Break %s\n",
         (getcbrk()) ? "Activat": "Dezactivat");
    setcbrk(0); // Il dezactiveaza
  }
```

***Observație:** Funcția **setcbrk** stabilește starea testării CTRL+BREAK pentru sistem, nu numai pentru programul curent. Atunci când programul se încheie, starea selectată anterior rămâne în vigoare. Amintiți-vă, veți lucra cu mesajele în Windows utilizând diferite comenzi, iar detectarea pentru CTRL+BREAK nu este utilă, în general, în programele Windows.*

556 POSIBILELE EFECTE SECUNDARE DIN DOS

C/C++

În secțiunea 555 ați învățat cum se utilizează funcția *setcbrk* pentru a schimba starea testării extinse pentru CTRL+BREAK. De asemenea, în capitolul despre discuri și fișiere, ați învățat cum se modifică starea verificării discurilor. Câteva alte secțiuni au prezentat modalități în care programele dumneavoastră pot schimba unitatea sau directorul curente. Atunci când programele dumneavoastră efectuează astfel de operații, ar trebui să salvați valorile inițiale când programul începe, astfel încât el să le poată restabili la încheierea sa.

Cu excepția cazurilor când programele își propun modificarea uneia sau mai multor astfel de valori, programele dumneavoastră nu ar trebui să le lase modificate după încheierea lor. Astfel de modificări ale valorilor sunt denumite *efecte secundare*, pe care ar trebui să le evitați. Atunci când un utilizator rulează programul său de contabilitate, de exemplu, unitatea și directorul implicite ale utilizatorului nu ar trebui schimbate după încheierea programului. De asemenea, chiar modificări mai subtile, cum ar fi dezactivarea verificării discului sau a verificării extinse pentru CTRL+BREAK, ar trebui să nu aibă loc. Când creați un program, includeți instrucțiunile suplimentare pe care sistemul dumneavoastră le solicită pentru a restabili valorile inițiale ale mediului.

Observație: *Amintiți-vă, veți lucra cu mesajele în Windows, utilizând diferite comenzi. Detectarea pentru CTRL+BREAK nu este utilă, în general, în programele Windows.*

557 SUSPENDAREA TEMPORARĂ A UNUI PROGRAM

C/C++

În capitolul despre dată și oră al acestei cărți, veți utiliza funcția *delay* pentru a opri programul pentru un număr specificat de milisecunde. În mod similar, programele dumneavoastră pot utiliza funcția *sleep* pentru a specifica intervalul de întrerupere în secunde, cum arătăm în continuare:

```
#include <dos.h>

void sleep(unsigned secunde);
```

Deoarece funcția *delay* lucrează cu milisecunde, ea este mai precisă decât funcția *sleep*. Totuși, puteți utiliza funcția *sleep* pentru a mări portabilitatea programului dumneavoastră la alt sistem de operare. Majoritatea sistemelor dispun de funcția *sleep*, care permite programelor să intre în stare inactivă până trece un anumit interval de timp sau apare un anumit eveniment. Următorul program, *sleep_5.c*, utilizează funcția *sleep* pentru o întrerupere de 5 secunde:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    printf("Pe punctul de a adormi pentru 5 secunde\n");
    sleep(5);
    printf("Desteptarea\n");
  }
```

Observaţie: *Să nu confundaţi comanda* ***sleep*** *din C cu comanda* ***sleep*** *din API Windows. Veţi învăţa mai multe despre comanda* ***sleep*** *din API Windows în capitolul despre procese şi fire de execuţie al acestei cărţi.*

SĂ NE AMUZĂM CU SUNETELE

C/C++ 558

În aproape toate PC-urile există un mic (de slabă calitate) *difuzor* pe care programele îl utilizează de obicei pentru a genera un sunet (*beep*). Folosind, însă, funcţia *sound,* de care dispun multe compilatoare de C, programele dumneavoastră pot genera sunete care emit pe diferite frecvenţe prin *difuzor.* Funcţia *sound* permite programelor să conecteze difuzorul pentru a emite un sunet de o anumită frecvenţă. Funcţia *nosound* deconectează difuzorul:

```
#include <dos.h>

void sound(unsigned frecventa);
void nosound(void);
```

Următorul program, *sirena.c,* utilizează funcţia *sound* pentru a genera un sunet de sirenă. La apăsarea oricărei taste, programul deconectează difuzorul, utilizând funcţia *nosound,* ca mai jos:

```
#include <dos.h>
#include <conio.h>

void main(void)
  {
    unsigned frecventa;
    do
      {
        for (frecventa = 500; frecventa <= 1000; frecventa += 50)
          {
            sound(frecventa);
            delay(50);
          }
        for (frecventa = 1000; frecventa >= 500; frecventa -= 50)
          {
            sound(frecventa);
            delay(50);
          }
      }
    while (! kbhit());
    nosound();
  }
```

OBŢINEREA INFORMAŢIILOR SPECIFICE DE ŢARĂ

C/C++ 559

Aşa cum ştiţi, ţări din lumea întreagă utilizează sistemul de operare DOS. Pentru a accepta utilizatori internaţionali, sistemul DOS acceptă diferite şabloane de tastatură, pagini de cod şi

informații specifice de țară. Pentru a ajuta programele dumneavoastră să determine valorile pentru țara curentă, programele pot utiliza funcția *country*:

```
#include <dos.h>
struct COUNTRY *country(int cod, struct COUNTRY *info);
```

Dacă reușește execuția, funcția *country* returnează un pointer la o structură de tip *COUNTRY*:

```
struct COUNTRY
  {
    int co_date;          // Formatul datei calendaristice
    char co_curr[5];      // Simbolul monetar
    char co_thsep[2];     // Separator pentru mii
    char co_desep[2];     // Separator pentru zecimale
    char co_dtsep[2];     // Separator pentru data
    char co_tmsep[2];     // Separator pentru timp
    char co_currstyle;    // Stilul monedei
    char co_digits;       // Cifre semnificative pentru moneda
    char co_time;         // Formatul timpului
    long co_case;         // Pointer catre macheta tastaturii
    char co_dasep;        // Separator pentru data
    char co_fill[10];     // Spatiu disponibil pentru alte
                          // caracteristici
  };
```

Valoarea *cod* specifică codul țării pe care doriți să o selectați. Dacă valoarea parametrului *info* este –1, funcția *country* va stabili codul țării curente la codul pe care îl specificați. Dacă valoarea lui *info* nu este –1, funcția *country* va atribui bufferului valorile pentru codul țării curente. Următorul program, *country.c*, afișează valorile țării curente:

```
#include <stdio.h>
#include <dos.h>
void main(void)
  {
    struct COUNTRY info;
    country(0, &info);
    if (info.co_date == 0)
     printf("Format data: luna/ziua/an\n");
    else if (info.co_date == 1)
     printf("Format data: ziua/luna/an\n");
    else if (info.co_date == 2)
     printf("Format data: an/luna/ziua\n");
    printf("Simbol monetar %s\n", info.co_curr);
    printf("Separator zecimal %s\n", info.co_thsep);
    printf("Separator data %s Separator timp %s\n",
      info.co_dtsep, info.co_tmsep);
    if (info.co_currstyle == 0)
```

```
    printf("Simbolul monetar precede valoarea fara spatii\n");
  else if (info.co_currstyle == 1)
    printf("Simbolul monetar urmeaza valoarea fara spatii\n");
  else if (info.co_currstyle == 2)
    printf("Simbolul monetar precede valoarea cu spatii\n");
  if (info.co_currstyle == 4)
    printf("Simbolul moneda urmeaza valoarea cu spatii\n");
  printf("Cifrele semnificative ale monedei %d\n",
     info.co_digits);
  if (info.co_time)
    printf("Sistem orar cu 24 ore\n");
  else
    printf("Sistem orar cu 12 ore\n");
  printf("Separatorul datei %s\n", info.co_dasep);
}
```

ADRESA DE TRANSFER PE DISC

C/C++ 560

Înainte de apariția versiunii DOS 3.0, programele efectuau operații cu fișiere utilizând blocurile de control al fișierelor (*file control blocks* – FCB). În mod implicit, atunci când DOS citește sau scrie informații, el o face prin intermediul unei zone de memorie numită *aria de transfer pe disc*. Aria de transfer pe disc este, în mod implicit, de 128 de octeți. Adresa primului octet al ariei este numită adresa de transfer pe disc (*disk transfer address* - DTA). În mod implicit, sistemul DOS utilizează deplasamentul 80H din prefixul segmentului de program ca adresă de transfer pe disc. Așa cum veți învăța în capitolul acestei cărți care tratează redirectarea I/O și procesarea liniei de comandă, deplasamentul de 80H al prefixului segmentului de program conține, de asemenea, și linia de comandă a programului. Deoarece majoritatea programelor nu utilizează operațiile de disc cu blocul de control al fișierelor, mulți programatori consideră că pot ignora adresa de transfer pe disc. Din păcate, rutine ca *findnext* și *findfirst*, prezentate în capitolul despre fișiere al acestei cărți, plasează rezultatele lor în adresa de transfer pe disc, suprascriind linia de comandă a programului dumneavoastră. Pentru a preveni ca operațiile care utilizează adresa de transfer pe disc să suprascrie parametrii liniei de comandă a programelor, mulți programatori utilizează un serviciu DOS pentru stabilirea adresei de transfer pe disc astfel încât să indice un buffer de memorie diferit. Așa cum veți învăța în secțiunea 561, programele dumneavoastră pot modifica și determina adresa de transfer pe disc utilizând funcțiile de bibliotecă *run-time*.

ACCESUL ȘI CONTROLUL ARIEI DE TRANSFER PE DISC

C/C++ 561

În secțiunea 560 ați învățat că aria de transfer pe disc este o zonă de 128 octeți pe care sistemul DOS o utilizează pentru servicii de I/O bazate pe blocul de control al fișierelor sau pentru operațiile *findnext* și *findfirst*. Pentru a vă ajuta să controlați aria de transfer pe disc, majoritatea compilatoarelor de C acceptă funcțiile *getdta* și *setdta*:

```
#include <dos.h>

char *far getdta(void);
void setdta(char far *adresa_transfer_disc);
```

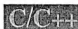
Funcția *getdta* returnează un pointer *far* (32 biți) la aria curentă de transfer pe disc. De asemenea, funcția *setdta* vă permite să atribuiți adresa de transfer pe disc a programului dumneavoastră la adresa *far* pe care o specificați. Următorul program, *dta.c*, ilustrează utilizarea funcțiilor *getdta* și *setdta*:

```
#include <stdio.h>
#include <dos.h>
#include <malloc.h>

void main(void)
  {
    char far *dta;
    dta = getdta();
    printf("DTA curenta este %lX\n", dta);
    if (MK_FP(_psp, 0x80) == dta)
      printf("DTA este la aceeasi locatie cu linia de comanda\n");
    dta = _fmalloc(128);
    setdta(dta);
    printf("Noua DTA este %lX\n", getdta());
  }
```

***Observație:** Când programați sub Windows, nu veți avea nevoie să controlați adresa de transfer pe disc, deoarece Windows utilizează modelul de memorie virtuală – detaliat în capitolul numit Gestiunea memoriei Windows – pentru a manipula majoritatea operațiilor de I/O cu fișiere.*

562 UTILIZAREA SERVICIILOR DE TASTATURĂ DIN BIOS



Atât DOS, cât și BIOS sau funcțiile de bibliotecă C run-time pun la dispoziție servicii care permit programelor dumneavoastră accesul la tastatură. De regulă, trebuie mai întâi să utilizați funcțiile de bibliotecă run-time a limbajului C. Dacă aceste funcții nu sunt adecvate, atunci utilizați funcțiile DOS. Dacă funcțiile DOS nu reușesc, încercați serviciile BIOS. Utilizarea funcțiilor de bibliotecă C run-time permite programelor dumneavoastră să se mențină mai portabile. Pentru a ajuta programele dumneavoastră să acceseze serviciile de tastatură din BIOS, biblioteca run-time de C dispune de funcția *_bios_keybrd*:

```
#include <bios.h>

unsigned _bios_keybrd(unsigned comanda);
```

Parametrul *comanda* specifică operația dorită. Tabelul 562 listează valorile posibile pe care le puteți transmite pentru parametrul *comanda*.

Valoarea	Semnificația	
_KEYBRD_READ	Indică funcției *_bios_keybrd* să citească un caracter de la bufferul tastaturii. Dacă octetul inferior al valorii returnate este 0, octetul superior conține un cod de tastatură extins.	
_KEYBRD_READY	Indică funcției *_bios_keybrd* să determine dacă este prezent un caracter în bufferul tastaturii. Dacă funcția *_bios_keybrd* returnează 0, înseamnă că nici o intrare de la tastatură nu este prezentă. Dacă valoarea returnată este 0xFFFF, utilizatorul a apăsat CTRL+C.	
_KEYBRD_SHIFTSTATUS	Indică funcției *_bios_keybrd* să returneze starea tastelor de control (*shift state*):	
	Bit 7	INS este activat
	Bit 6	CAPSLOCK este activat
	Bit 5	NUMLOCK este activat
	Bit 4	SCROLLOCK este activat
	Bit 3	Tasta ALT este apăsată
	Bit 2	Tasta CTRL este apăsată
	Bit 1	Tasta SHIFT stânga este apăsată
	Bit 0	Tasta SHIFT dreapta este apăsată
_NKEYBRD_READ	Indică funcției *_bios_keybrd* să citească un caracter din bufferul tastaturii. Dacă octetul inferior al valorii returnate este 0, octetul superior conține un cod extins al tastaturii. _NKEYBRD_READ indică funcției *_bios_keybrd* să citească tastele speciale, cum ar fi tastele cu săgeți.	
_NKEYBRD_READY	Indică funcției *_bios_keybrd* să determine dacă este prezent un caracter în bufferul tastaturii. Dacă funcția *_bios_keybrd* returnează 0, nu este prezentă nici o intrare de la tastatură. Dacă valoarea returnată este 0xFFFF, utilizatorul a apăsat CTRL+C. Valoarea _NKEYBRD_READY indică funcției *_bios_keybrd* să accepte tastele speciale, cum ar fi tastele cu săgeți.	
_NKEYBRD_SHIFTSTATUS	Indică funcției *_bios_keybrd* să returneze starea tastelor de control ale tastaturii, inclusiv a tastelor speciale.	
	Bit 15	SYSREQ este apăsată
	Bit 14	CAPSLOCK este apăsată
	Bit 13	NUMLOCK este apăsată
	Bit 12	SCROLLOCK este apăsată
	Bit 11	Tasta ALT dreapta este apăsată
	Bit 10	Tasta CTRL dreapta este apăsată
	Bit 9	Tasta ALT stânga este apăsată
	Bit 8	Tasta CTRL stânga este apăsată

Tabelul 562 *Valorile posibile pentru parametrul* ***comanda.***

Următorul program, *keystare.c*, utilizează o buclă pentru a afişa schimbările de stare ale tastaturii până la apăsarea oricărei taste în afara tastelor SHIFT, ALT, CTRL, NUMLOCK şi aşa mai departe. Programul citeşte numai tastele care nu sunt speciale, după ca mai jos:

```
#include <stdio.h>
#include <bios.h>

void main(void)
  {
    unsigned int stare, stare_veche = 0;
    do
      {
        stare = _bios_keybrd(_KEYBRD_SHIFTSTATUS);
        if (stare != stare_veche)
          {
            stare_veche = stare;
            if (stare & 0x80)
              printf("Tasta Ins activata");
            if (stare & 0x40)
              printf("Tasta Caps activata");
            if (stare & 0x20)
              printf("Tasta Num Lock activata");
            if (stare & 0x10)
              printf("Tasta Scroll Lock activata");
            if (stare & 0x08)
              printf("Tasta Alt apasata");
            if (stare & 0x04)
              printf("Tasta Ctrl apasata");
            if (stare & 0x02)
              printf("Tasta Shift stanga apasata");
            if (stare & 0x01)
              printf("Tasta Shift dreapta apasata");
            printf("\n");
          }
      }
    while (! _bios_keybrd(_KEYBRD_READY));
  }
```

Observaţie: *Multe compilatoare de C dispun de o funcţie numită* ***bioskey*** *care execută procesări similare cu funcţia* ***_bios_keybrd.*** *Studiaţi documentaţia care însoţeşte compilatorul pentru a determina ce funcţie acceptă.*

Observaţie: *Windows nu acceptă comanda* ***_bios_keybrd*** *sau echivalentele ei. Sub Windows, veţi obţine informaţii despre tastatură utilizând diferite clase specifice fiecărui compilator. De exemplu, în mediul de compilare Borland* ***C++ 5.02****, veţi determina dacă tasta CAPS LOCK sau altă tastă de acest gen este selectată, cu ajutorul proprietăţii* ***KeyboardFlags*** *a clasei* ***KeyboardManager.*** *Consultaţi documentaţia compilatorului pentru mai multe informaţii despre urmărirea informaţiilor despre tastatură în Windows.*

OBȚINEREA LISTEI CU ECHIPAMENTE DIN BIOS

C/C++563

Pe măsură ce crește complexitatea programelor dumneavoastră, uneori va trebui ca ele s determine caracteristici ale componentei hardware a calculatorului. În aceste cazuı programele dumneavoastră pot utiliza funcția *_bios_equiplist*:

```
#include <bios.h>
unsigned _bios_equiplist(void);
```

Funcția *_bios_equiplist* returnează o valoare de tip *unsigned int*, ai cărei biți au următoare semnificație:

```
struct Equip
{
    unsigned floppy_available:1;
      // 1 daca e prezenta unitatea de floppy-disc
    unsigned coprocessor_available:1;
      // 1 daca e prezent coprocesorul matematic
    unsigned system_memory:2;
      // memoria RAM
    unsigned video_mode:2;
     // 01 mod video 40 x 25 mono
     // 10 mod video 80 x 25 color
     // 11 mod video 80 x 25 mono
    unsigned floppy_disk_count:2;
     // Adauga 1 pentru fiecare unitate de floppy noua
    unsigned serial_port_count:2;
    unsigned game_adapter_available:1;
     // 1 daca e prezent adaptorul de jocuri
    unsigned printer_count:2;
};
```

Următorul program, *echip.c*, utilizează funcția *_bios_equiplist* pentru a afișa lista echipamentelor sistemului:

```
#include <stdio.h>
#include <bios.h>
void main(void)
  {
    struct Equip
      {
        unsigned floppy_available:1;
        unsigned coprocessor_available:1;
        unsigned system_memory:2;
        unsigned video_memory:2;
        unsigned floppy_disk_count:2;
        unsigned unused_1:1;
```

```
      unsigned serial_port_count:3;
      unsigned game_adapter_available:1;
      unsigned unused_2:1;
      unsigned printer_count:2;
    };
  union Equipment
    {
      unsigned lista;
      struct Equip biti_lista;
    } echip;
  echip.lista = _bios_equiplist();
  if (echip.biti_lista.coprocessor_available)
    printf("Coprocesor matematic disponibil\n");
  else
    printf("Nu exista coprocesor matematic\n");
  printf("Memoria placii de baza %d\n",
     (echip.biti_lista.system_memory + 1) * 16);
  printf("Numarul de unitati floppy %d\n",
     echip.biti_lista.floppy_disk_count + 1);
  printf("Numarul de imprimante %d\n",
     echip.biti_lista.printer_count);
  printf("Numarul de porturi seriale %d\n",
     echip.biti_lista.serial_port_count);
}
```

*Observație: Unele compilatoare de C dispun de funcția **biosequip** care execută procese similare cu funcția **_bios_equiplist**. Acceptarea funcției **_bios_equiplist** și a comenzilor corelate sub Windows diferă de la compilator la compilator. De exemplu, Borland **C++ 5.02** acceptă comanda **_bios_equiplist,** în timp ce Visual C++ utilizează apelări ale interfeței API Windows pentru a obține informații despre sistem. Studiați documentația compilatorului dumneavoastră pentru detalii. Compact discul care însoțește această carte conține programul **Win_Equip.cpp** care utilizează interfața Win API pentru a returna informații despre sistem.*

564 CONTROLUL INTRĂRILOR ȘI IEȘIRILOR PENTRU PORTUL SERIAL



Pentru a ajuta programele dumneavoastră să execute operații de I/O la portul serial, cum ar fi COM1, multe compilatoare de mediu DOS dispun de funcția *bioscom* (sau *_bios_serialcom*):

```
#include <bios.h>

unsigned bioscom(int comanda, int port, char octet);
```

Parametrul *comanda* specifică operația pe care o doriți și trebuie să fie o valoare listată în tabelul 564.1.

Valoare	Descriere
_COM_INIT	Stabileşte valorile pentru comunicare ale portului
_COM_RECEIVE	Primeşte un octet de la port
_COM_SEND	Trimite un octet la port
_COM_STATUS	Returnează valorile portului

Tabelul 564.1 *Valorile posibile pentru parametrul* ***comanda****.*

Parametrul *port* specifică portul serial pe care îl doriţi, unde 0 corespunde lui COM1, 1 lui COM2 şi aşa mai departe. Parametrul *octet* specifică fie octetul pentru ieşire, fie valorile de comunicare pe care le doriţi. Dacă octetul conţine valorile de comunicare pe care le doriţi, el poate conţine o combinaţie a valorilor listate în tabelul 564.2.

Valoare	Descriere
_COM_CHR7	Date pe 7 biţi
_COM_CHR8	Date pe 8 biţi
_COM_STOP1	1 bit de stop
_COM_STOP2	2 biţi de stop
_COM_NOPARITY	Fără paritate
_COM_ODDPARITY	Paritate impară
_COM_EVENPARITY	Paritate pară
_COM_110	110 baud
_COM_150	150 baud
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1200 baud
_COM_2400	2400 baud
_COM_4800	4800 baud
_COM_9600	9600 baud

Tabelul 564.2 *Valorile posibile pentru parametrul* ***octet****.*

Spre deosebire de comandă, cel mai semnificativ octet al valorii returnate are una dintre semnificaţiile listate în tabelul 564.3.

Biţi	Semnificaţie
8	Datele sunt pregătite
9	Eroare de depăşire
10	Eroare de paritate
11	Eroare de încadrare
12	Detectează întrerupere
13	Registrul de memorare a transferului e gol

(continuare

Biți	Semnificație
14	Registrul de deplasare a transferului e gol
15	Pauză

***Tabelul 564.3** Semnificația biților valorii returnate de funcția **_bios_serialcom**.*

Pentru *_COM_INIT și _COM_STATUS,* funcția *_bios_serialcom* definește cel mai puțin semnificativ octet al valorii returnate în funcție de valorile din tabelul 564.4.

Biți	Semnificație când are valoarea 1
0	Schimbă în gata pentru transmisie
1	Schimbă în date pregătite
2	Detectează frontul constant al semnalului de apel
3	Schimbă starea în detector de semnal de linie de recepție
4	Gata de transmisie
5	Datele pregătite
6	Indicator de apel
7	Semnal de linie detectat

Tabelul 564.4** Valorile returnate când se utilizează **_COM_INIT** și **_COM_STATUS.

Următorul program, *setcom1.c*, stabilește datele de comunicare pentru COM1 la 9600 baud, date pe 8 biți, 1 bit de stop și fără paritate:

```
#include <stdio.h>
#include <bios.h>

void main(void)
  {
    char i = 0, titlu[] = "Jamsa\'s C/C++ Programmer\'s Bible";
    unsigned stare;
    stare = _bios_serialcom(_COM_INIT, 0, _COM_9600 | _COM_CHR8
       | _COM_STOP1 | _COM_NOPARITY);

    if (stare & 0x100) // Datele sunt pregatite
      while (titlu[i])
        {
          _bios_serialcom(_COM_SEND, 0, titlu[i]);
          putchar(titlu[i]);
          i++;
        }
  }
```

***Observație**: Unele compilatoare dispun de o funcție **bioscom** care execută procese similare. Compact discul care însoțește această carte include programul **win_serial.cpp** care citește portul serial din cadrul programelor în Windows.*

ACCESUL LA SERVICIILE DOS CU AJUTORUL FUNCȚIEI BDOS

C/C++565

Așa cum ați învățat, funcția *intdos* permite programelor dumneavoastră să acceseze serviciile DOS. Unele dintre serviciile DOS utilizează numai registrele AX și DX. Pentru astfel de servicii programele pot folosi funcția *bdos*:

```
#include <dos.h>

int bdos(int functie_dos, unsigned registru_dx,
  unsigned registru_al);
```

Parametrul *functie_dos* indică serviciul apelat. Parametrii registru_dx și registru_al specifică valorile pe care le așteaptă serviciul în registrele DX și AL. Funcția returnează valoarea registrului AX la terminarea serviciului. Următorul program, *bdos.c*, utilizează funcția *bdos* pentru a afișa unitatea de disc curentă:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    int unitate;
    unitate = bdos(0x19, 0, 0);
    printf("Unitatea de disc curenta este %c\n", 'A' + unitate);
  }
```

Observație: *Funcția **bdos** transmite o valoare **unsigned** pentru registrul DX. Dacă utilizați serviciile DOS care cer un pointer, puteți utiliza funcția **bdosptr**. Dacă utilizați modelul de memorie small, al doilea parametru va corespunde lui DX. În cazul modelului de memorie large, valoarea va corespunde lui DS:DX.*

OBȚINEREA DE INFORMAȚII EXTINSE DESPRE ERORI ÎN DOS

C/C++566

Atunci când un serviciu al sistemului DOS eșuează, programele dumneavoastră pot cere informația suplimentară despre acea eroare, pentru a determina sursa și cauza erorii. Pentru a vă ajuta să obțineți informații extinse despre eroare, multe compilatoare de C pun la dispoziție funcția *dosexterr*, ca mai jos:

```
#include<dos.h>

int dosexterr(struct DOSERROR *info_eroare);
```

Parametrul *info_eroare* este un pointer la o structură de tipul *DOSERROR* care conține informații extinse despre eroare, ca mai jos:

```
struct DOSERROR
  {
    int de_exterror;      // eroarea extinsa
    int de_class;         // clasa erorii
    int de_action;        // actiunea recomandata
```

```
    int de_locus;          // sursa erorii
};
```

Dacă funcția *dosexterr* returnează 0, înseamnă că precedentul apel al unui serviciu DOS nu a avut nici o eroare. Valoarea de eroare extinsă indică o eroare specifică. Clasa erorii descrie categoria erorii, după cum arătăm în tabelul 566.1.

Valoarea	Semnificația
01H	Resurse depășite
02H	Eroare temporară
03H	Eroare de autorizare
04H	Eroare de sistem
05H	Eroare de hardware
06H	Eroare de sistem nedatorată programului curent
07H	Eroare de aplicație
08H	Articol neîntâlnit
09H	Format nevalid
0AH	Articol blocat
0BH	Eroare de suport
0CH	Articolul există
0DH	Eroare necunoscută

Tabelul 566.1 *Clasele de erori pe care le returnează* ***dosexterr*** *în cadrul membrului* ***de_class.***

Membrul *de_action* (acțiunea recomandată) indică programului cum să răspundă erorii, după cum arătăm în tabelul 566.2.

Valoarea	Semnificația
01H	Mai întâi încearcă din nou, apoi cere intervenția utilizatorului.
02H	Încearcă din nou, cu o întârziere, apoi cere intervenția utilizatorului.
03H	Cere intervenția utilizatorului pentru soluție.
04H	Renunță și elimină.
05H	Renunță, dar nu elimină.
06H	Ignoră eroarea.
07H	Încearcă din nou după intervenția utilizatorului.

Tabelul 566.2 *Valori posibile returnate de membrul* ***de_action***.

În sfârșit, membrul *de_locus* specifică sursa erorii, după cum arată tabelul 566.3.

Valoarea	Semnificația
01H	Sursă necunoscută
02H	Eroare de dispozitiv bloc
03H	Eroare de rețea

Valoarea	Semnificaţia
04H	Eroare de dispozitiv serial
05H	Eroare de memorie

***Tabelul 566.3** Valorile returnate de membrul **de_locus**.*

Atunci când programele dumneavoastră trebuie să răspundă erorilor într-o modalitate îngrijită şi studiată, trebuie să utilizaţi structura *dosexterr* pentru a obţine informaţii suplimentare.

DETERMINAREA VOLUMULUI DE MEMORIE CONVENŢIONALĂ BIOS

C/C++ 567

Multe dintre programele mai vechi nu beneficiază de avantajele memoriei extinse şi expandate. Ele utilizează numai memoria convenţională de 640Kb a PC-ului. Atunci când examinaţi astfel de programe, puteţi întâlni apelarea funcţiei *biosmemory*, care returnează cantitatea de memorie convenţională (în Kb) raportată de către BIOS la pornirea sistemului. Cantitatea de memorie returnată de *biosmemory* nu cuprinde memoria extinsă, expandată sau memoria superioară. Veţi implementa funcţia *biosmemory* ca mai jos:

```
#include<bios.h>

int biosmemory(void);
```

În plus faţă de funcţia *biosmemory*, puteţi întâlni funcţia *_bios_memsize*, care efectuează o procesare identică. Veţi implementa *_bios_memsize* ca mai jos:

```
#include<bios.h>

int _bios_memsize(void);
```

Următorul program, *biosmem.c*, va afişa volumul de memorie pe care BIOS îl raportează ca răspuns la invocarea funcţiilor *biosmemory* şi *_bios_memsize*:

```
#include <stdio.h>
#include <bios.h>

void main(void)
  {
    printf("BIOS raporteaza o memorie de %dKb\n", biosmemory());
    printf("BIOS raporteaza o memorie de %dKb\n", _bios_memsize());
  }
```

***Observaţie:** Deoarece modelul de memorie din Windows utilizează memoria virtuală, trebuie să utilizaţi apelurile interfeţei Windows API, explicate în capitolul despre gestionarea memoriei în Windows, atunci când scrieţi programe pentru Windows.*

CONSTRUIREA POINTERILOR FAR

C/C++ 568

Un pointer *far* constă într-un segment de 16 biţi şi o adresă de deplasament de 16 biţi. Atunci când lucraţi cu pointeri *far*, puteţi să împărţiţi pointerul în segmentul şi deplasamentul său. De asemenea, puteţi să construiţi un pointer *far* dintr-un segment şi o adresă de deplasa-

ment. Pentru a vă ajuta să construiți un pointer *far*, compilatorul de C dispune de funcția macro *MK_FP*:

```
#include <dos.h>
void far *MK_FP(unsigned segment, unsigned deplasament);
```

Următorul fragment de cod utilizează funcția macro *MK_FP* pentru a construi un pointer *far* încolo de adresa unei variabile *near*:

```
long far *fptr;
long variabila;
struct SREG segs;

    // Obtine segmentul curent de date
segread(&segs);
fptr = MK_FP(segs.ds, &variabila);
```

Pentru a înțelege mai bine funcția macro *MK_FP*, să studiem următoarea implementare:

```
#define MK_FP(s, o) ((void far *)(((long) s << 16) | (0)))
```

Pentru a crea o adresă *far* de 32 biți, funcția macro creează o valoare *long* și deplasează biții adresei segmentului către cei 16 biți superiori ai valorii. Apoi, funcția macro utilizează o operație *SAU* pe biți pentru a atribui adresa de deplasament celor 16 biți inferiori.

***Observație:** Așa cum ați învățat, deoarece pointerii **far** nu se aplică modelului de memorie virtuală, programele Windows nu îi utilizează.*

569 ÎMPĂRȚIREA UNEI ADRESE FAR ÎN SEGMENT ȘI DEPLASAMENT



Așa cum am arătat în secțiunea 568, un pointer *far* constă într-un segment de 16 biți și o adresă de deplasament de 16 biți. Atunci când lucrați cu pointeri *far*, puteți să împărțiți adresa pe care o referențiază un pointer *far*, în segmentul și deplasamentul corespunzător. În astfel de cazuri, programele dumneavoastră pot utiliza funcțiile macro *FP_SEG* și *FP_OFF*:

```
#include <dos.h>
unsigned FP_OFF(void far *pointer);
unsigned FP_SEG(void far *pointer);
```

Următoarea instrucțiune ilustrează utilizarea funcțiilor macro *FP_SEG* și *FP_OFF*:

```
char far *titlu = "Totul despre C/C++";
unsigned segment, deplasament;
segment = FP_SEG(titlu);
deplasament = FP_OFF(titlu);
```

***Observație:** Așa cum ați învățat, deoarece pointerii **far** nu se aplică modelului de memorie virtuală, programele Windows nu îi utilizează.*

DETERMINAREA MEMORIEI LIBERE

C/C++ 570

Atunci când programele dumneavoastră alocă memorie, puteți să utilizați funcția *coreleft* pentru a estima volumul de memorie convențională disponibilă la acel moment pentru alocare. Funcția *coreleft* nu oferă un raport exact al memoriei neutilizate. În schimb, dacă utilizați modelul de memorie small, funcția *coreleft* returnează memoria neutilizată dintre vârful memoriei heap și stivă. Dacă utilizați modelul de memorie *large*, funcția *coreleft* returnează cantitatea de memorie dintre vârful memoriei alocate și sfârșitul memoriei convenționale. Funcția *coreleft* returnează memoria neutilizată în octeți. În cazul modelului de memorie small, funcția *coreleft* returnează o valoare *unsigned*:

```
#include <alloc.h>
unsigned coreleft(void);
```

Dacă utilizați modelul de memorie *large*, funcția *coreleft* va returna o valoare de tip *long*:

```
#include <alloc.h>
long coreleft(void);
```

Următorul program, *coreleft.c*, afișează volumul de memorie disponibilă. Programul utilizează constantele modelului de memorie pe care multe compilatoare le acceptă pentru a determina modelul curent de memorie:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
#if defined(__SMALL__)
    unsigned rezultat;
#else
    long rezultat;
#endif
    rezultat = coreleft();
    printf("Volumul de memorie disponibilă este %dKb\n",
      rezultat / 1024);
  }
```

Observație: *Dacă compilatorul nu acceptă funcția* ***coreleft****, verificați dacă dispune de funcțiile* ***_memavl*** *și* ***_memmax****. Capitolul despre gestionarea memoriei în Windows detaliază modul în care se determină memoria disponibilă în cadrul mediului Windows.*

CITIREA VALORILOR REGISTRULUI SEGMENT

C/C++ 571

Atunci când lucrați în mediu DOS, compilatorul va urmări codul programului dumneavoastră, datele și stiva utilizând patru registre segment. Cele patru registre segment sunt listate în tabelul 571.

Numele	Descrierea
CS	Registru segment de cod
DS	Registru segment de date
SS	Registru segment de stivă
ES	Registru segment de extra

Tabelul 571 *Cele patru registre segment din sistemul DOS.*

În funcție de modelul de memorie al programelor dumneavoastră, fiecare registru segment poate indica un unic segment de 64Kb sau două sau mai multe registre segment pot indica același segment. Când programele dumneavoastră utilizează serviciile DOS și BIOS, uneori trebuie să cunoașteți valoarea registrului segment. Pentru astfel de cazuri, puteți utiliza funcția *segread*:

```
#include <dos.h>
void segread(struct SREGS *segs);
```

Fișierul antet *dos.h* definește structura *SREGS*:

```
struct SREGS
   {
     unsigned int es;
     unsigned int cs;
     unsigned int ss;
     unsigned int ds;
   };
```

Următorul program, *sregs.c*, utilizează funcția *segread* pentru a afișa conținutul registrului segment curent:

```
#include <stdio.h>
#include <dos.h>

void main(void)
   {
     struct SREGS segs;
     segread(&segs);
     printf("CS %X DS %X SS %X ES %X\n", segs.cs, segs.ds,
       segs.ss, segs.es);
   }
```

572 TIPURILE DE MEMORIE



PC-ul poate să conțină trei tipuri de memorie: convențională, extinsă și expandată. Câteva dintre secțiunile care urmează abordează în detaliu aceste tipuri de memorie. Pe măsură ce programați, este important să înțelegeți diferitele tipuri de memorie și caracteristicile lor. Pașii pe care trebuie să-i faceți pentru a aloca și utiliza diferitele tipuri de memorie vor diferi. În plus, fiecare tip de memorie are viteze diferite de acces, ceea ce afectează performanța programelor dumneavoastră. Pentru a determina volumul și tipul de memorie instalată în

PC-ul dumneavoastră, puteți să utilizați comanda DOS 5 (sau versiuni mai noi) MEM /CLASSIFY, ca mai jos:

```
C:\> MEM /CLASSIFY <ENTER>
```

Dacă nu utilizați DOS 5 sau alte versiuni mai noi, trebuie să actualizați sistemul dumneavoastră. DOS 5 dispune de câteva capacități de gestionare a memoriei care vă ajută să maximizați utilizarea memoriei PC-ului dumneavoastră.

MEMORIA CONVENȚIONALĂ

C/C++ 573

Când IBM a pus în funcțiune primul PC în 1981, calculatorul utiliza, de obicei, între 64Kb și 256Kb de RAM. Atunci, această memorie era mai mult decât suficientă. Această memorie a devenit cunoscută ca *memorie convențională* a PC-ului. Astăzi, memoria convențională a unui PC este primul 1Mb de RAM. Programele DOS rulează, în mod obișnuit, cu primii 640 Kb de memorie convențională. PC-ul utilizează 384Kb de memorie (numită *rezervată* sau *memorie superioară*) care se situează între 640Kb și 1Mb pentru memoria video a calculatorului, driverele de dispozitive, alte dispozitive hardware mapate în memorie și BIOS. Ani întregi, însă, sistemele de operare nu au utilizat secțiuni extinse din această memorie rezervată. Începând cu versiunea 5, sistemul DOS dispune de modalități în care programele dumneavoastră și driverele de dispozitive se pot situa în zone neutilizate, atunci când programele rulează. Avantajele memoriei superioare vă permit să eliberați mai mult decât 640Kb de memorie convențională pentru uzul sistemului DOS. Pentru informații în legătură cu modul în care se beneficiază de avantajele zonei de memorie superioare, studiați în documentația DOS intrarea din CONFIG.SYS DOS=UMB (*upper memory block* - blocul de memorie superioară).

Așa cum ați învățat, Windows utilizează modelul de memorie virtuală pentru a gestiona memoria, ceea ce înseamnă că eliberarea memoriei convenționale nu are semnificație sub Windows. Însă, memoria convențională este importantă când rulați programe în cadrul unei ferestre DOS sub Windows.

MACHETA MEMORIEI CONVENȚIONALE

C/C++ 574

În secțiunea 573 ați învățat că *memoria convențională* este primul 1Mb de RAM al calculatorului dumneavoastră. Programele dumneavoastră și DOS se află de obicei în primii 640Kb de memorie convențională. Pentru a vă ajuta să înțelegeți mai bine cum utilizează DOS memoria convențională, figura 574 prezintă macheta memoriei convenționale.

Capitolul despre DOS și BIOS al acestei cărți explică vectorii de întrerupere BIOS și zona de comunicare BIOS. Nucleul DOS este acel software, *io.sys* și *msdos.sys*, pe care DOS îl încarcă în memorie la pornirea sistemului. Intrările *config.sys* reprezintă regiunea de memorie pe care DOS o alocă pentru driverele dispozitivelor, bufferele de disc și așa mai departe. Zonele de memorie pentru *comand.com rezident și temporar* păstrează acel software responsabil pentru afișarea *promptului DOS* și prelucrarea comenzilor pe care le introduceți. Pentru a face ca programele dumneavoastră să dispună de mai multă memorie, DOS împarte *comand.com* într-o secțiune rezidentă, care rămâne permanent în memorie și o secțiune temporară, pe care o poate suprascrie fiecare comandă. După ce comanda se execută, porțiunea rezidentă din *comand.com* reîncarcă secțiunea temporară de pe disc. Cei 384Kb de memorie dintre 640Kb și 1MB reprezintă memoria superioară a calculatorului dumneavoastră, care conține memoria video, blocurile de memorie superioară și serviciile BIOS bazate pe ROM, așa cum am explicat în capitolul despre DOS și BIOS al acestei cărți.

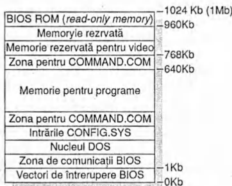


Figura 574 *Harta memoriei convenționale a PC-ului.*

Observație: *Considerațiile despre memoria convențională nu sunt atât de importante pentru Windows 95 și Windows NT, cum sunt în DOS. Deoarece Windows utilizează modelul de memorie virtuală, majoritatea execuțiilor programelor dumneavoastră va avea loc în afara primului 1Mb de RAM - fie în locațiile RAM mai înalte, fie în memoria virtuală pe care calculatorul o utilizează pe unitatea de hard-disc. Secțiunile ulterioare vor aborda modelul memoriei virtuale în detaliu.*

575 ACCESUL LA MEMORIA CONVENȚIONALĂ



Pe scurt, *modelul memoriei* programului dumneavoastră definește modul de utilizare a memoriei convenționale a programului. În funcție de modelul de memorie pe care îl utilizează programul dumneavoastră, compilatorul va aloca unul sau mai multe segmente de 64Kb pentru stocarea codului programului și datele sale. Atunci când programul dumneavoastră trebuie să aloce memorie dinamic, puteți utiliza funcțiile C, cum ar fi *malloc*, pentru a aloca memorie din zona de memorie de manevră (*heap*) *near* sau *_fmalloc*, pentru a aloca memorie din zona de memorie de manevră *far*. Secțiunile 597 și 598 vor aborda zonele de memorie de manevră *near* și *far*. În plus, programele dumneavoastră pot utiliza serviciile sistemului DOS pentru alocarea memoriei.

Observație: *Ca regulă, programele dumneavoastră trebuie să utilizeze numai o metodă de alocare și dealocare a memoriei. Pentru a mări portabilitatea programelor dumneavoastră, acestea trebuie să încerce să utilizeze funcțiile de bibliotecă C run-time pentru gestionarea memoriei. Nu combinați funcțiile C de alocare a memoriei cu cele furnizate de DOS. Prin combinarea funcțiilor C și DOS de alocare a memoriei, măriți posibilitatea erorilor și faceți ca programul dumneavoastră să devină mai dificil de înțeles.*

576 SĂ ÎNȚELEGEM DE CE PC-UL ȘI SISTEMUL DOS SUNT LIMITATE LA 1MB



De multe ori se face referire la *bariera de 640Kb*, atunci când se discută despre DOS. Pe scurt, bariera de 640Kb se referă la regiunea memoriei convenționale în cadrul căreia programele dumneavoastră trebuie să ruleze. Așa cum ați învățat, însă, programele DOS

utilizează, de fapt, serviciile BIOS și memoria video, care se situează în intervalul de memorie dintre 640Kb și 1Mb. În plus, începând cu DOS 5, programele dumneavoastră și driverele de dispozitive pot să se situeze în zona de memorie superioară, astfel că restricțiile de memorie DOS apar de fapt la 1Mb.

Limita de 1Mb memorie este mai mult o limită a PC-ului decât o limită a sistemului DOS. PC-ul inițial (care utiliza procesorul 8088) a utilizat o adresă de segment pe 16 biți și un deplasament (*offset*) pe 16 biți în cadrul segmentului. În interiorul memoriei PC-ului, segmentarea are loc la interval de 16 octeți. Cele 65536 de adrese unice de segment permit PC-ului să acceseze 65536 * 16 octeți (sau 1 048 576) locații de memorie unice. Deoarece sistemul DOS trebuie să ruleze în acest mediu, el este incorect blamat pentru restricționarea memoriei programului dumneavoastră.

Observație: *Windows utilizează un tip special pe 32 de biți, cunoscut ca* ***DWORD*** *pentru stocarea adreselor de segment și deplasament. Cei 32 de biți din* ***DWORD*** *permit accesul Windows până la 4Gb de RAM, disponibili dacă procesorul calculatorului este capabil de a accesa atâta memorie. Majoritatea calculatoarelor Pentium pot accesa până la 128Mb de RAM.*

PRODUCEREA UNEI ADRESE DIN SEGMENTE ȘI DEPLASAMENTE

C/C++ 577

Pentru a gestiona locațiile de memorie destinate adreselor, PC-ul utilizează o adresă de *segment* și de *deplasament* pe 16 biți. Adresa de segment identifică în mod obișnuit începutul unei regiuni de 64Kb. Adresa de deplasament identifică un octet în cadrul regiunii. Segmentele pot începe la intervale de 16 octeți, numite *paragrafe*. Pentru a adresa memoria, PC-ul combină adresa de segment și de deplasament pentru a produce o adresă de 20 de biți, care poate să adreseze 1048576 locații unice de memorie (1Mb). Pentru a crea adresa de 20 de biți, PC-ul deplasează la stânga adresa de segment de 16 biți cu patru locații de biți și apoi adaugă la rezultat adresa de deplasament. De exemplu, să presupunem că adresa de segment este 1234H. Când PC-ul deplasează adresa la stânga, rezultatul devine 12340H. Apoi, dacă adresa de deplasament este 5, rezultatul este 12340H + 5H sau 12345H. Următoarea ecuație ilustrează mai bine procesul implicat:

```
1234H Segment deplasat devine 12340H
Adunat deplasamentul de 0005H
Rezulta 12345H
```

Dacă examinați operația în binar, rezultatul devine următorul:

```
     0001 0010 0011 0100          Segment
0001 0010 0011 0100 0000          Deplasat
                    0101          Deplasament
========================
0001 0010 0011 0100 0001          Rezultat (adresa de 20 biti)
```

MEMORIA EXPANDATĂ

C/C++ 578

Așa cum ați învățat, programele DOS rulează în mod obișnuit, în cadrul memoriei convenționale de 640Kb a calculatorului. Multe dintre programele mai mari, însă, cum ar fi foile de calcul, cer mai mult de 640Kb. PC-ul IBM inițial (8088) nu putea să adreseze

memorie mai mult de 1Mb. Pentru a permite PC-ului accesul la mai mult de 1Mb de memorie, companiile Lotus, Intel și Microsoft au creat o specificație *pentru memoria expandată* (EMS), care combină software și o platformă specială de memorie expandată pentru a „păcăli" PC-ul în scopul accesării unor volume mari de memorie.

Pentru a utiliza memoria expandată, calculatorul dumneavoastră trebuie să conțină o platformă de memorie expandată. Pentru început, un sofware al specificației pentru memorie expandată alocă un bloc de 64Kb în cadrul memoriei superioare (regiunea de 384Kb dintre 640Kb și 1Mb). Apoi, acel software împarte regiunea de 64Kb în patru secțiuni de 16 Kb, numite *pagini*. Când programul începe, el utilizează funcțiile specificației pentru memorie expandată pentru a aloca și încărca regiunea de memorie expandată. Pentru a face aceasta, programul dumneavoastră definește pagini logice (16Kb) în cadrul regiunii memoriei expandate.

De exemplu, dacă aveți o foaie de calcul de 128Kb, calculatorul împarte datele în opt pagini logice de 16Kb. Când programul trebuie să acceseze o anumită pagină logică, el utilizează o funcție a specificației pentru memorie expandată pentru *a mapa* pagina logică într-una din paginile specificației pentru memorie expandată în memoria superioară calculatorul dumneavoastră, pe care programul DOS poate apoi să o acceseze direct. Cum programele dumneavoastră utilizează alte pagini logice, el mapează paginile în interiorul și în afara zonei specificației de memorie expandată, după cum este necesar.

Multe programe DOS cer maparea memoriei expandate numai pentru că procesorul 8088 nu poate accesa locațiile de memorie dincolo de 1Mb. Deși memoria expandată oferă procesorului 8088 calea de a accesa volume mari de date, repetata mapare de date provoacă o considerabilă suprasarcină, care scade performanța sistemului dumneavoastră. Dacă utilizați un procesor 80286 sau mai mare, calculatorul dumneavoastră poate să acceseze memoria dincolo de 1Mb (numită memorie extinsă), care este un proces mult mai rapid.

Observație: *Atunci când un program utilizează memoria expandată, codul programului rămâne în regiunea de memorie convențională de 640Kb. Numai datele programelor se pot situa în zona de memorie expandată.*

Observație: *Așa cum ați învățat în secțiunea 574, considerațiile despre memoria convențională nu sunt atât de importante în Windows 95 și Windows NT, cum sunt în DOS. Deoarece memoria convențională nu este așa importantă, memoria expandată este, de asemenea, mai puțin importantă pentru calculatoarele mai noi, indiferent dacă lucrează sub Windows sau DOS. De fapt, procesoarele mai noi (x486 sau superioare) nu mai acceptă memoria expandată, ci numai memoria extinsă, cum detaliază secțiunea 580.*

579 *UTILIZAREA MEMORIEI EXPANDATE*



Secțiunea 578 prezintă utilizarea memoriei expandate. De regulă, pentru a mări performanța, programele dumneavoastră ar trebui să folosească *memoria extinsă* (vezi secțiunea 580) în locul memoriei expandate. Însă, dacă împrejurările vă obligă să scrieți un program care trebuie să ruleze pe un PC mai vechi, cu procesor 8088, programele dumneavoastră vor accesa serviciile specificației pentru memorie expandată utilizând funcția *int86* și INT 67H, așa cum detaliază capitolul despre DOS și BIOS al acestei cărți. Există multe servicii diferite ale specificației pentru memorie expandată care vă permit alocarea, maparea, dealocarea și manipularea memoriei expandate. Compact discul care însoțește această carte conține un exemplu de program, *ems.c*, care ilustrează modul cum puteți utiliza memoria expandată în

cadrul programelor dumneavoastră. Următoarea funcție, *test_ems*, utilizează registrele de memorie și operații pe biți pentru a determina dacă a fost încărcat driverul pentru memoria expandată în calculatorul dumneavoastră:

```
int test_ems(void)
  {
     union REGS inregs, outregs;
     struct SREGS segs;
     int major, minor;      // versiune DOS
     struct DeviceHeader  {
      struct DeviceHeader far *link;
      unsigned attributes;
      unsigned strategy_offset;
      unsigned interrupt_offset;
      char name_or_number_of_units[8];
     } far *dev;
     int i;
     char nume_ driver[9];
     // Obtine versiunea DOS
     inregs.x.ax = 0x3001;
     intdos (&inregs, &outregs);
     major = outregs.h.al;
     minor = outregs.h.ah;
     if (major < 2)
      return(0); // Necesita DOS 2.0
     else
      {
     // Obtine lista de liste
     inregs.h.ah = 0x52;
     intdosx (&inregs, &outregs, &segs);
     if (major == 2)
       dev = (struct DeviceHeader far *)
        MK_FP(segs.es + 1, outregs.x.bx + 7);
     else if ((major == 3) && (minor == 0))
       dev = (struct DeviceHeader far *)
        MK_FP(segs.es + 2, outregs.x.bx + 8);
     else
       dev = (struct DeviceHeader far *)
        MK_FP(segs.es + 2, outregs.x.bx + 2);
     while (FP_OFF(dev) != 0xFFFF)
      {
        if (dev->attributes & 0x8000)
          {
// Dispozitiv de tip caracter
          for (i = 0; i < 8; i++)
           nume_driver [i] = dev-> name_or_number_of_units[i];
          nume_driver[8] = NULL;
```

```
      }
      if (! strcmp(nume_driver, "EMMXXXX0"))
        return(1); // A gasit driverul
      dev = dev->link;
      }
    }
  return(0);
}
```

580 MEMORIA EXTINSĂ



'C-ul IBM inițial (8088) utiliza adresarea de 20 biți, care restricționa accesarea memoriei la Mb. Începând cu IBM AT (80286), PC-ul căpătă capacitatea de a utiliza adresarea de 24 biți, are îi permite adresarea până la 16Mb. Mașinile bazate pe 386, 486, 586 și 686 măresc dresarea la 32 biți, ceea ce permite PC-ului adresarea până la 4Gb de memorie. Atunci când rimul PC a căpătat posibilitatea de adresare dincolo de 1Mb, programatorii au numit nemoria de peste 1Mb *memorie extinsă*. Deoarece PC-ul inițial nu putea accesa memoria lincolo de 1Mb, el nu putea utiliza memoria extinsă.

'entru a accesa memoria extinsă, trebuie să încărcați un driver de dispozitiv pentru memoria xtinsă. În sistemul DOS, driverul este, de obicei, ***himem.sys***. Atunci când programele lumneavoastră din mediul DOS utilizează memoria extinsă, numai datele programului pot fi ocalizate în memoria extinsă. Codul programului trebuie să rămână în zona de 640Kb a nemoriei convenționale. Când programele dumneavoastră utilizează memoria extinsă, însă, erviciile de sistem care permit accesul la memorie trebuie să schimbe modul de execuție entru CPU, de la modul real la modul protejat și apoi invers. Schimbarea modurilor CPU ere timp de procesare, introducând astfel o suprasarcină. Suprasarcina este, totuși, mai mică lecât în cazul memoriei expandate – ceea ce face memoria extinsă mai bună.

)bservație: *Așa cum ați învățat în secțiunea 574, considerațiile despre memoria convenională nu sunt atât de importante în Windows 95 și Windows NT cum sunt în DOS. Calculatoarele care rulează Windows 95 utilizează încă o așa-numită „memorie extinsă" entru a gestiona accesul la memorie de dincolo de primul 1Mb al calculatorului. Calculaoarele care rulează Windows NT gestionează memoria fără utilizarea evidentă a memoriei xtinse cerută de Windows 95. Așa cum veți învăța în secțiunile următoare, programele lumneavoastră ar trebui să utilizeze modelul de memorie virtuală pentru a gestiona orice ip de memorie din programele Windows.*

581 MODUL REAL ȘI MODUL PROTEJAT



istemul DOS este un sistem de operare cu sarcină unică (*single-tasking*), ceea ce înseamnă ă în mod obișnuit rulează un singur program la un moment dat (cu excepția driverelor de lispozitive și a programelor rezidente în memorie). Deoarece sistemul DOS este un sistem le operare single-tasking, protejarea unui program de un altul nu este o problemă emnificativă. De aceea, sistemul DOS permite programelor dumneavoastră accesarea nemoriei în orice fel doresc ele. Cu alte cuvinte, programele în DOS pot modifica orice aloare a locațiilor de memorie convențională. Atunci când rulați mai multe programe în celași timp, un program nu poate modifica memoria aleatoriu, așa cum poate în cazul

mediului single-tasking, deoarece ar fi posibil ca programul să suprascrie conținutul unui alt program din memorie. Într-un mediu multi-program, sistemul de operare trebuie să protejeze memoria unui program de a altuia. Pentru a proteja programele din memorie, sistemul de operare se bazează pe protejarea memoriei prin hardware.

Începând cu cipul 80286, CPU poate rula în unul sau două moduri: *real* sau *protejat*. Modul real există pentru compatibilitatea cu primul PC IBM, bazat pe procesor 8088. Sistemul DOS utilizează modul real, care nu are capacitatea de protecție a memoriei. Alte sisteme de operare, cum ar fi Unix, OS/2 sau Windows, pot rula în modul protejat. În modul protejat, un program nu poate accesa memoria altui program. În plus, în cadrul modului protejat, PC-ul modifică schema sa de adresare cu segment și deplasament, cu una care permite ca CPU să utilizeze adresarea cu 24 biți în cazul procesorului 80286 și adresarea cu 32 biți în cazul mașinilor cu 386 sau superioare. În acest fel, modul protejat permite PC-ului adresarea la memoria extinsă pentru cod și date.

Atunci când programele în DOS utilizează memoria extinsă, folosesc un software pentru accesarea memoriei extinse care comută în mod evident unitatea CPU de la modul real (în care rulează DOS) la modul protejat (care poate accesa memoria extinsă) și apoi comută din nou, la modul real.

ACCESUL LA MEMORIA EXTINSĂ — C/C++ 582

Înainte ca programele dumneavoastră să poată utiliza memoria extinsă, trebuie să instalați un driver de dispozitiv pentru memoria extinsă (de obicei, *himem.sys*). Apoi, utilizând întreruperea multiplexă DOS, serviciu 4300H la INT 2FH, programele dumneavoastră pot obține un punct de intrare în memoria calculatorului pentru serviciile memoriei extinse. Driverul pentru memoria extinsă dispune de funcții care permit programelor dumneavoastră să aloce, să dealoce și să manipuleze memoria extinsă. Pentru a accesa serviciile, atribuiți diferiți parametri registrelor PC-ului și apoi branșați la punctul de intrare specificat. Pentru a vă ajuta să înțelegeți mai bine serviciile memoriei extinse, compact discul care însoțește această carte include fișierul *xmsdemo.c*.

ZONA DE MEMORIE ÎNALTĂ — C/C++ 583

Așa cum ați învățat, memoria extinsă este memoria de peste 1Mb a calculatorului. Atunci când programele DOS accesează memoria extinsă, CPU schimbă modul real cu modul protejat și apoi invers. Dacă utilizați un 386 și DOS 5 sau versiuni mai noi, veți putea beneficia de avantajele unei „scăpări" la proiectarea procesorului 386 care vă permite să accesați primii 64Kb ai memoriei extinse din modul real. Așa cum arătăm în figura 583, această zonă de 64Kb este numită zona de memorie înaltă.

Cea mai bună modalitate de utilizare a zonei de memorie înaltă este încărcarea nucleului DOS în această zonă, eliberând memoria din cadrul celor 640Kb de memorie convențională. Dacă însă DOS nu utilizează zona de memorie înaltă, programul dumneavoastră o poate aloca utilizând serviciul memoriei extinse. Pentru a încărca sistemul DOS în zona de memorie înaltă, trebuie să instalați driverul *himem.sys* și apoi să utilizați intrarea DOS=HIGH în fișierul *config.sys*.

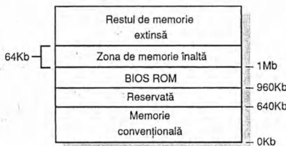


Figura 583 *Zona de memorie înaltă reprezintă primii 64Kb ai memoriei extinse.*

Observație: *Așa cum arată această secțiune, în programele dumneavoastră ar trebui să nu utilizați zona de memorie înaltă dacă ele vor rula pe un calculator mai recent decât x386. Dacă programele dumneavoastră rulează sub Windows 95 sau Windows NT, programele nu ar trebui să utilizeze zona de memorie înaltă.*

584 STIVA



Stiva (stack) este o regiune de memorie în cadrul căreia programele păstrează temporar datele pe durata execuției. De exemplu, atunci când programele dumneavoastră transmit parametri către funcție, C plasează parametrii în stivă. Când funcția își încheie execuția, C îi scoate din stivă. De asemenea, atunci când funcțiile dumneavoastră declară variabile locale, C păstrează valorile variabilelor în stivă pe durata execuției funcției. Când funcția își încheie execuția, C elimină variabilele.

Stiva este astfel numită pentru că programele *depun* (*push*) valorile în stivă, la fel cum se depun tăvile una peste alta într-o cafenea și apoi se *extrag* (*pop*) din stivă, la fel cum se iau tăvile din cafenea, de sus în jos. În funcție de modelul de memorie al programului, volumul spațiului în stivă pus la dispoziție de compilator va diferi. În funcție de cum folosește programul dumneavoastră funcțiile și parametrii, volumul de stivă necesar programului va diferi. Ca valoare minimă, compilatorul va aloca 4Kb de spațiu în stivă. Dacă programul dumneavoastră necesită mai mult sau mai puțin spațiu în stivă, puteți utiliza directivele compilatorului și editorului de legături pentru a controla volumul de spațiu în stivă alocat de compilator și de editorul de legături. PC-ul utilizează două registre pentru a localiza stiva. Registrul segment de stivă (SS – *stack segment*) indică începutul stivei, iar registrul pointer de stivă (SP – *stack pointer*) indică vârful stivei.

585 DIFERITE CONFIGURAȚII ALE STIVEI



În secțiunea 584 ați învățat că programul utilizează stiva pentru a stoca temporar informații în primul rând pe durata apelării funcțiilor. În funcție de utilizarea funcțiilor în program și de numărul și dimensiunea parametrilor pe care programul îi transmite acestor funcții, volumul de spațiu în stivă cerut de program va diferi de la un program la altul. Când utilizați modelul de memorie *small*, C va aloca spațiu în stivă începând de la vârful segmentului de date, ca în figura 585.

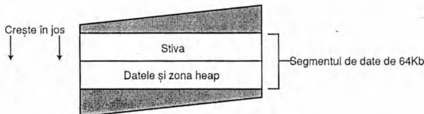


Figura 585 *Modelul C de memorie* ***small*** *de alocare a spațiului în stivă.*

Pe de altă parte, când utilizați modelele de memorie *large* sau *compact*, C va aloca pentru stivă un întreg segment de 64Kb. Dacă programul plasează mai multe informații în stivă decât poate reține aceasta, va apărea o eroare de depășire a stivei (*stack-overflow*). Dacă programul a dezactivat testarea stivei, nu veți lua la cunoștință eroarea, iar datele depuse în stivă pot suprascrie datele din program. Secțiunea 586 prezintă modalități prin care puteți determina dimensiunea curentă a stivei pentru program.

Observație: *Veți învăța mai mult despre testarea stivei în capitolul despre optimizare al acestei cărți.*

Windows, însă, construiește stiva în mod diferit față de stiva DOS. În Windows volumul implicit de spațiu pentru stivă este de 1Mb, iar limita dimensiunii sale este limita memoriei virtuale, ceea ce înseamnă că stiva poate fi chiar mai mare de 250Mb. În consecință, dimensiunea stivei de 250Mb, reduce preocuparea dumneavoastră pentru protejarea programului la erori de depășire a stivei.

DETERMINAREA DIMENSIUNII CURENTE A STIVEI

C/C++ 586

În funcție de utilizarea funcțiilor și parametrilor programului dumneavoastră, volumul de spațiu pentru stivă cerut de program va diferi. Utilizând directivele compilatorului și ale editorului de legături, programele dumneavoastră pot aloca o anumită dimensiune pentru stivă. Atunci când programele dumneavoastră se execută, puteți să știți dimensiunea curentă a stivei. Dacă folosiți Turbo C++ Lite, puteți utiliza variabila globală *_stklen*. Următorul program, *stklen.c*, utilizează variabila globală *_stklen* pentru a afișa dimensiunea curentă a stivei:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    printf("Dimensiunea curenta a stivei %d octeti\n", _stklen);
  }
```

Dacă folosiți Microsoft Visual C++, funcția *stackavail* returnează volumul de spațiu disponibil pentru stivă.

587 CONTROLUL SPAȚIULUI STIVEI CU VARIABILA GLOBALĂ _STKLEN

C/C++

În secțiunea 586 ați învățat că variabila globală *_stklen* permite programelor dumneavoastră determinarea dimensiunii curente a stivei. În plus, programele dumneavoastră pot utiliza variabila globală *_stklen* pentru a controla volumul de spațiu pentru stivă alocat de compilator. Pentru a specifica dimensiunea stivei cu variabila *_stklen*, programele dumneavoastră trebuie să o declare ca o variabilă globală externă. Următorul program, *stiva8kb.c*, utilizează variabila globală *_stklen* pentru a aloca o stivă de 8Kb:

```
#include <stdio.h>
#include <dos.h>

extern unsigned _stklen = 8096;

void main(void)
  {
    printf("Dimensiunea curenta a stivei %d octeti\n", _stklen);
  }
```

***Observație:** În Windows, majoritatea compilatoarelor vă vor permite stabilirea dimensiunii stivei pe care o cere un anumit fir de execuție în cadrul comenzii care creează respectivul fir. Capitolul acestei cărți, intitulat „Procese și fire de execuție" discută în detaliu despre firele de execuție.*

588 ATRIBUIREA UNEI VALORI LA UN INTERVAL DE MEMORIE

C/C++

Atunci când programele dumneavoastră lucrează cu matrice și pointeri la intervale de memorie, puteți să inițializați memoria la o anumită valoare. Pentru a face aceasta, programele dumneavoastră pot utiliza funcția *memset*. Veți implementa funcția *memset* ca mai jos:

```
#include <mem.h>
void *memset(void *ptr, int caracter, size_t num_octeti);
```

Parametrul *ptr* este un pointer la primul octet din intervalul de memorie. Parametrul *caracter* este valoarea octetului pe care doriți să o atribuiți intervalului de memorie. În sfârșit, parametrul *num_octeti* specifică numărul de octeți din intervalul de memorie. Funcția returnează un pointer la începutul intervalului de memorie. Următoarea instrucțiune utilizează funcția *memset* pentru a inițializa o matrice de șiruri de caractere cu *NULL*:

```
char sir[128];
memset(sir, NULL, sizeof(sir));
```

589 COPIEREA UNUI INTERVAL DE MEMORIE ÎN ALTUL

C/C++

Când programele dumneavoastră lucrează cu șiruri de caractere, ele pot utiliza funcția *strcpy* pentru a copia conținutul unui șir de caractere în altul. Atunci când însă trebuie să copiați o

matrice de valori întregi sau în virgulă mobilă, programele dumneavoastră pot executa prelucrare similară, utilizând funcțiile *memmove* sau *memcpy*:

```
#include <mem.h>

void *memmove(void *destinatie, const void *sursa,
  size_t num_octeti);
void *memcpy(void *destinatie, const void *sursa,
  size_t num_octeti);
```

Parametrii *destinatie* și *sursa* sunt pointeri la o matrice în care funcția copiază date (*destinatie*) și de la care funcția face copia (*sursa*). Parametrul *num_octeti* specifică număr de octeți de copiat. Principala diferență dintre cele două funcții este că funcția *memmo* copiază corect datele dintre două intervale de octeți care se pot suprapune în memorie, timp ce funcția *memcpy* poate copia incorect datele. Următorul program, *memmove* utilizează funcția *memmove* pentru a copia conținutul unei matrice de valori în virgu mobilă:

```
#include <stdio.h>
#include <mem.h>

void main(void)
  {
    float val[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    float vid[5];
    int i;
    memmove(vid, val, sizeof(val));
    for (i = 0; i < 5; i++)
      printf("%3.1f ", vid[i]);
  }
```

COPIEREA UNUI INTERVAL DE MEMORIE PÂNĂ LA UN ANUMIT OCTET

C/C++ 59

Atunci când programele dumneavoastră lucrează cu matrice, va trebui uneori să copi conținutul unei matrice la altă matrice. În funcție de conținutul matricei, copierea se po face până la *n* octeți sau să se încheie imediat ce întâlnește un anumit caracter. Pentru executa o astfel de procesare, programele dumneavoastră pot utiliza funcția *memccpy*:

```
#include <mem.h>

void *memccpy(void *destinatie, const void *sursa,
  int caracter, size_t num_octeti);
```

Parametrii *destinatie* și *sursa* sunt pointeri la matricea în care funcția copiază dat (*destinatie*) și de la care funcția face copia (*sursa*). Parametrul *caracter* conține caracte care, dacă este copiat, operația de copiere se încheie imediat. Parametrul *num_oc* specifică numărul de octeți de copiat. Dacă funcția copiază *num_octeti*, ea returne valoarea *NULL*. Dacă funcția întâlnește caracterul pe care l-ați specificat, funcția returnea un pointer la octetul din destinație care urmează imediat după caracterul specific

Următorul program, *memccpy.c*, utilizează funcția *memccpy* pentru a copia literele de la A la K în matricea destinație:

```
#include <stdio.h>
#include <mem.h>

void main(void)
  {
    char alfabet[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    char destinatie[27];
    char *rezultat;
    rezultat = memccpy(destinatie, alfabet, 'K', sizeof(alfabet));
    if (rezultat)
      *rezultat = NULL;
    printf(destinatie);
  }
```

591 COMPARAREA A DOUĂ MATRICE DE TIP UNSIGNED CHAR



Atunci când programele dumneavoastră lucrează cu matrice, puteți să comparați două intervale de memorie. Cea mai obișnuită comparație între două intervale de memorie este verificarea a două șiruri de caractere. Pentru a compara două intervale de memorie, programele dumneavoastră pot utiliza funcțiile *memcmp* și *memicmp*, ca mai jos:

```
#include <mem.h>

void *memcmp(const void *bloc_1, const void *bloc_2,
  size_t num_octeti);
void *memicmp(const void *bloc_1, const void *bloc_2,
  size_t num_octeti);
```

Diferența dintre funcțiile *memcmp* și *memicmp* este că funcția *memicmp* ignoră existența majusculelor și minusculelor. Parametrii *bloc_1* și *bloc_2* sunt pointeri la începutul fiecărui interval de octeți. Parametrul *num_octeti* specifică numărul de octeți de comparat. Funcția returnează valorile listate în tabelul 591.

Valoare	Semnificație
mai mică decât 0	*bloc_1* este mai mic decât *bloc_2*
0	Blocurile sunt la fel
mai mare decât 0	*bloc_1* este mai mare decât *bloc_2*

Tabelul 591 *Valorile returnate de funcțiile* ***memcmp*** *și* ***memicmp.***

Următorul program, *memcmp.c*, utilizează funcțiile *memcmp* și *memicmp* pentru a compara două șiruri de caractere :

```
#include <stdio.h>
#include <mem.h>

void main(void)
```

```
{
   char *a = "AAA";
   char *b = "BBB";
   char *c = "aaa";

   printf("Compara %s cu %s, cu functia memcmp %d\n", a, b,
          memcmp(a, b, sizeof(a)));
   printf("Compara %s cu %s, cu functia memicmp %d\n", a, c,
          memicmp(a, c, sizeof(a)));
}
```

INTERSCHIMBAREA OCTEȚILOR ADIACENȚI DIN ȘIRURI DE CARACTERE

C/C++ 592

Când lucrați pe diferite tipuri de calculatoare, puteți să interschimbați octeți adiacenți de memorie. Pentru a face aceasta, programele dumneavoastră pot utiliza funcția *swab*:

```
#include <stdlib.h>

void swab(char *sursa, char *destinatie, int num_octeti);
```

Parametrul *sursă* este un pointer la un șir de caractere ai cărui octeți vreți să îi interschimbați. Parametrul *destinatie* este un pointer la un șir de caractere la care funcția *swab* atribuie octeții interschimbați. Parametrul *num_octeti* specifică numărul de octeți de interschimbat. Următorul program, *swab.c*, ilustrează funcția *swab*:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mem.h>

void main(void)
  {
    char *sursa = "aJsm\'a s/C+C +rPgoarmmres\'B biele";
    char destinatie[64];

    memset(destinatie, NULL, sizeof(destinatie));
    swab(sursa, destinatie, strlen(sursa));
    printf("Sursa: %s Destinatie %s\n", sursa, destinatie);
  }
```

ALOCAREA MEMORIEI DINAMICE

C/C++ 593

Atunci când programele dumneavoastră declară o matrice, compilatorul de C alocă memorie pentru a păstra matricea. Dacă trebuie să modificați cerințele programului și matricea trebuie să se mărească sau să se micșoreze, trebuie să modificați și să recompilați programul. Pentru a reduce numărul de modificări pe care trebuie să le aduceți programului pentru schimbarea dimensiunilor matricei, programele dumneavoastră pot aloca propria lor memorie pe durata execuției. Când alocați memorie în această modalitate, biblioteca run-time de C returnează

un pointer la începutul intervalului de memorie. Programele pot apoi lucra cu memoria, utilizând un format matrice sau pointer, după cum preferați. Atunci când alocați memorie pe durata execuției, programul dumneavoastră poate utiliza funcția de bibliotecă run-time *malloc*:

```
#include <alloc.h>
void *malloc(size_t nr_octeti);
```

Parametrul *nr_octeti* specifică numărul de octeți pe care îi doriți pentru dimensiunea matricei. Dacă funcția *malloc* reușește alocarea intervalului de octeți, ea va returna un pointer la începutul intervalului. Dacă apare o eroare, funcția *malloc* va returna *NULL*. Următorul program, *malloc.c*, utilizează funcția *malloc* pentru a aloca memorie pentru o matrice de șiruri de caractere, o matrice de valori întregi și o matrice de valori în virgulă mobilă:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *sir;
    int *val_int;
    float * val_float;
    if ((sir = (char *) malloc(50)))
      printf("Alocare reusita sir de 50 octeti\n");
    else
      printf("Eroare la alocarea sir\n");
    if ((val_int = (int *) malloc(100 * sizeof(int))) != NULL)
      printf("Alocare reusita val_int[100]\n");
    else
      printf("Eroare la alocarea val_int[100]\n");
    if ((val_float = (float *) malloc(25 * sizeof(float))) != NULL)
      printf("Alocare reusita val_float[25]\n");
    else
      printf("Eroare la alocarea val_float[25]\n");
  }
```

După cum vedeți, programul invocă funcția *malloc* cu numărul cerut de octeți. Dacă funcția *malloc* returnează *NULL*, programul va afișa un mesaj de eroare.

594 DIN NOU DESPRE CONVERSIE



În secțiunea 593 ați învățat că puteți utiliza funcția de bibliotecă run-time C *malloc* pentru a spune programelor dumneavoastră să aloce memorie pe durata execuției. Așa cum ați învățat, funcția *malloc* returnează un pointer *void*:

```
void *malloc(size_t nr_octeti);
```

Când utilizați funcția *malloc* pentru a aloca memorie, programele dumneavoastră pot converti rezultatul funcției *malloc* la un pointer de orice tip doriți. De exemplu, următoarea instrucțiune utilizează funcția *malloc* pentru a aloca un pointer la 100 de valori de tip *int*:

```
int *val_int;
val_int = (int *) malloc(100 * sizeof(int));
```

Dacă alocaţi memorie pentru a păstra 50 de valori în virgulă mobilă, instrucţiunea dumneavoastră va deveni următoarea:

```
int *val_float;
val_float = (float *) malloc(50 * sizeof(float));
```

Atunci când convertiţi astfel valorile returnate de funcţia *malloc*, puteţi elimina mesajele de avertizare ale compilatorului.

ELIBERAREA MEMORIEI CÂND NU MAI ESTE NECESARĂ

C/C++ 595

Aşa cum aţi învăţat, programele dumneavoastră pot utiliza funcţia *malloc* pentru a aloca memorie pe parcursul execuţiei, pentru a păstra matrice sau alte elemente. Când programele dumneavoastră nu mai au nevoie de memorie, ele pot elibera memoria, astfel ca programele dumneavoastră să o poată reutiliza în alte scopuri. Pentru a elibera memoria alocată, programele dumneavoastră pot utiliza funcţia *free*, ca mai jos:

```
#include <alloc.h>

void free(void *prt);
```

Parametrul *ptr* este un pointer la începutul intervalului de memorie pe care vreţi să o eliberaţi. Următorul program, *free.c*, utilizează funcţia *malloc* pentru a aloca o matrice de întregi. Programul utilizează apoi matricea. Când programul nu mai are nevoie de matrice, el utilizează funcţia *free* pentru a elibera memoria care corespunde matricei, ca mai jos:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    int *val_int;
    int i;
    if ((val_int = malloc(100 * sizeof(int))) == NULL)
      printf("Eroare la alocarea matricei\n");
    else
      {
        for (i = 0; i < 100; i++)
          val_int[i] = i;
        for (i = 0; i < 100; i++)
          printf("%d ", val_int[i]);
        free(val_int);
      }
  }
```

Observaţie: *Dacă programele dumneavoastră nu utilizează funcţia* ***free*** *pentru a elibera memoria, programul va elibera automat memoria la încheierea lui. De regulă, însă, programele ar trebui să elibereze memoria cât mai repede după ce nu mai e nevoie de ea.*

596 ALOCAREA MEMORIEI UTILIZÂND FUNCȚIA CALLOC

C/C++

Așa cum ați învățat, programele dumneavoastră pot utiliza funcția *malloc* pentru a aloca memorie dinamică pe durata execuției programelor. Când utilizați funcția *malloc*, specificați numărul de octeți pe care vreți să îi alocați. În plus față de utilizarea funcției *malloc*, C permite programelor dumneavoastră să aloce memorie utilizând funcția *calloc*. Diferența dintre cele două funcții este că funcția *malloc* vă cere să specificați numărul de octeți doriți, în timp ce funcția *calloc* vă cere să specificați numărul de elemente de o anumită dimensiune, pe care le doriți:

```
#include <alloc.h>

void *calloc(size_t nr_elemente, size_t dim_elemente);
```

Parametrul *nr_elemente* specifică pentru cât de multe elemente trebuie să aloce memorie funcția *calloc*. Parametrul *dim_elemente* specifică dimensiunea fiecărui element în octeți. Dacă funcția *calloc* reușește să aloce memoria, ea va returna un pointer la începutul intervalului de memorie. Dacă apare o eroare, funcția *calloc* va returna *NULL*. Următorul program, *calloc.c*, utilizează funcția *calloc* pentru a aloca tipuri diferite de matrice:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *sir;
    int *val_int;
    float *val_float;

    if ((sir = (char *) calloc(50, sizeof(char))))
     printf("Alocare reusita sir de 50 octeti\n");
    else
     printf("Eroare la alocarea sir\n");
    if ((val_int = (int *) calloc(100, sizeof(int))) != NULL)
     printf("Alocare reusita val_int[100]\n");
    else
     printf("Eroare la alocarea val_int[100]\n");
    if ((val_float = (float *) calloc(25, sizeof(float))) != NULL)
     printf("Alocare reusita val_float[25]\n");
    else
     printf("Eroare la alocarea val_float[25]\n");
  }
```

Observație: *Când programele dumneavoastră au încheiat utilizarea memoriei alocate de funcția* ***calloc****, programele trebuie să utilizeze funcția* ***free*** *pentru eliberarea memoriei.*

597 ZONA HEAP

C/C++

Atunci când programele dumneavoastră alocă memorie dinamică, biblioteca run-time de C obține memoria dintr-o zonă de memorie neutilizată numită *heap*. Atunci când compilați

programe, utilizând modelul de memorie *small*, zona *heap* este zona de memorie dintr vârful zonei datelor programului și stivă, cum arătăm în figura 597.

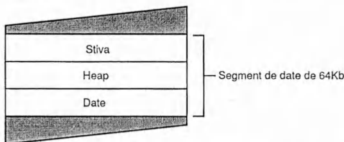


Figura 597 *Zona heap se situează între zona de date ale programului și stivă.*

După cum puteți vedea, zona *heap* se situează în segmentul de date al programului. D aceea, volumul de spațiu *heap* disponibil pentru programul dumneavoastră este fix pentr acest program, dar poate diferi de la un program la altul. Când utilizați funcțiile *malloc* sa *calloc* pentru a aloca memorie, cea mai mare memorie pe care funcțiile o pot aloca este d 64Kb (presupunând că zona *heap* nu conține date și stive). Următorul program, *nuspatiu.* încearcă să aloce trei matrice de 30Kb. Deoarece zona *heap* nu are 90Kb disponibil alocarea memoriei eșuează, ca mai jos:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *unu, *doi, *trei;

    if ((unu = (char *) malloc(30000)) == NULL)
      printf("Eroare la alocarea matricei unu\n");
    else if ((doi = (char *) malloc(30000)) == NULL)
      printf("Eroare la alocarea matricei doi\n");
    else if ((trei = (char *) malloc(30000)) == NULL)
      printf("Eroare la alocarea matricei trei\n");
    else
      printf("Toate matricele au fost alocate\n");
  }
```

În modelul de memorie *large*, dimensiunea totală a zonei *heap* nu este restricționată la 64Kl totuși, valoarea cea mai mare pe care o puteți aloca în orice moment este încă restricționat la segmentul de 64Kb. Pentru alocarea unor valori mai mari decât 64Kb, trebuie utiliza modelul de memorie *huge*. Încercați să compilați programul *nuspatiu.c* utilizând modelul d memorie *large*. Programul trebuie să fie capabil să satisfacă cerințele de alocare a memoriei.

Programele Windows utilizează zona *heap* în mod similar cu programele DOS. Totuși programele Windows au acces la două zone *heap*: *zona heap globală* și *zona heap local* Toate programele pot utiliza zona *heap* globală, pe care Windows o utilizează pentru indica blocurile de memorie mari (de 256 de octeți sau mai mult). De asemenea, Window

dă fiecărui program acces la propria sa zonă *heap* locală, pe care Windows o utilizează pentru a indica blocurile mici de memorie (256 octeți sau mai puțin). De regulă, majoritatea programelor Windows operează cu zona *heap* locală - deoarece nu are efectiv limite de dimensiune. Totuși, programele dumneavoastră pot utiliza zona *heap* locală pentru stocări reduse de memorie.

598 OCOLIREA LIMITEI DE 64KB A ZONEI HEAP



Așa cum ați învățat, când programele dumneavoastră DOS alocă memorie din zona *heap*, ele pot aloca cel mult 64Kb de memorie. Deoarece limita de 64Kb este o restricție DOS (PC în modul real), multe compilatoare din mediul DOS dispun de funcții numite *farmalloc* și *farcalloc* care permit programelor dumneavoastră să aloce memorie din zona *heap far*, care se situează în afara segmentului de date, ca mai jos:

```
#include <alloc.h>

void far *farcalloc(unsigned long nr_elemente, unsigned long
   dim_elemente);
void far *farmalloc(unsigned long nr_octeti);
```

Parametrii pe care programul dumneavoastră îi transmite funcțiilor *farcalloc* și *farmalloc* sunt identici cu cei transmiși funcțiilor *calloc* și *malloc*. Atunci când alocați memorie din zona *heap far*, veți utiliza un pointer *far* la datele programului. Următorul program, *fmalloc.c*, alocă matrice în zona *heap far*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char far *sir;
    int far *val_int;
    float far *int_float;

    if ((sir = (char *) farmalloc(50)))
      printf("Alocare reusita sir de 50 octeti\n");
    else
      printf("Eroare la alocarea sir\n");
    if ((val_int = (int *) farmalloc(100 * sizeof(int))) != NULL)
      printf("Alocare reusita val_int[100]\n");
    else
      printf("Eroare la alocarea val_int[100]\n");
    if ((val_float = (float *) farmalloc(25 * sizeof(float)))
       != NULL)
      printf("Alocare reusita val_float[25]\n");
    else
      printf("Eroare la alocarea val_float[25]\n");
  }
```

În cazul modelului de memorie *large*, programul tratează toți pointerii ca pointeri *far*. Totuși, e bine să utilizați explicit pointerul *far* în cadrul aplicației dumneavoastră.

Observație: *Atunci când programele utilizează funcțiile **farcalloc** sau **farmalloc** pentru a aloca memorie din zona heap far, ar trebui să utilizați funcția **farfree** pentru a elibera memoria când programele nu au nevoie de ea. Dacă utilizați un alt compilator, numele acestor funcții pot diferi. Consultați descrierea **zonei heap far** din documentația compilatorului dumneavoastră.*

În Windows, veți „ocoli" restricția de dimensiune a zonei *heap* prin alocarea din zona *heap* globală și nu din zona *heap* locală. De fapt, majoritatea compilatoarelor alocă în mod implicit din zona *heap* globală și dispun de diferite comenzi utilizate pentru alocarea din zona *heap* locală.

ALOCAREA MEMORIEI DIN STIVĂ C/C++ 599

Așa cum ați învățat, funcțiile *malloc* și *calloc* vă permit alocarea din zona *heap*. Când ați terminat cu utilizarea memoriei, ea trebuie eliberată cu ajutorul funcției *free*. În funcție de programul dumneavoastră, uneori trebuie să alocați memorie care să existe numai pe durata apelării unei anumite funcții. Pentru aceasta, programele dumneavoastră pot utiliza funcția *alloca* pentru alocarea memoriei din stivă, ca mai jos:

```
#include <malloc.h>

void *alloca(size_t nr_octeti);
```

Parametrul *nr_octeti* specifică dimensiunea intervalului de memorie pe care programul trebuie să îl aloce. Dacă funcția *alloca* reușește, ea va returna un pointer la începutul blocului de memorie. Dacă apare o eroare, funcția va returna *NULL*. Nu utilizați funcția *free* pentru eliberarea memoriei alocată de program utilizând funcția *alloca* – funcția *free* lucrează cu zona *heap*, în timp ce funcția *alloca* lucrează cu stiva. Programul eliberează memoria alocată în mod automat, când funcția care conține memoria alocată se încheie.

Observație: *Pentru ca programele să restabilească corect pointerul de stivă, funcția trebuie să conțină variabile locale. Pentru a asigura un cadru corect al stivei, declarați o variabilă locală după declararea variabilei pointer la care funcția **alloca** atribuie rezultatele, ca mai jos:*

```
char *pointer;
char stack_fix[1];

stack_fix[0] = NULL;
pointer = alloca(dim);
```

Următorul program, *alloca.c*, ilustrează cum se utilizează funcția *alloca*:

```
#include <stdio.h>
#include <malloc.h>

void o_functie(size_t dim)
  {
    int i;
```

```
    char *pointer;
    char stack_fix[1];
    stack_fix[0] = NULL;
    if ((pointer = alloca(dim)) == NULL)
      printf("Eroare la alocarea %u octeti din stiva\n", dim);
    else
      {
        for (i = 0; i < dim; i++)
          pointer[i] = i;
        printf("A alocat si utilizat un buffer de %u octeti\n", dim);
      }
  }

void main(void)
  {
    o_functie(1000);
    o_functie(32000);
    o_functie(65000);
  }
```

600 ALOCAREA DATELOR DE MARI DIMENSIUNI



Așa cum ați învățat, dimensiunea maximă a unei matrice create este de 64Kb. Dacă aplicația dumneavoastră necesită o matrice mai mare, puteți aloca memorie pentru o matrice *huge*. Pentru ca programele dumneavoastră să lucreze cu structuri de date de mari dimensiuni, multe compilatoare C de mediu DOS, dispun de funcțiile *halloc* și *hfree*:

```
#include <malloc.h>

void huge *halloc(long nr_elemente, size_t dim);
void hfree(void huge *pointer);
```

Parametrul *nr_elemente* specifică numărul de elemente ale matricei. Parametrul *dim* specifică dimensiunea fiecărui element în octeți. Dacă funcția *halloc* reușește, ea va returna un pointer la începutul zonei de memorie. Dacă apare o eroare, funcția *halloc* va returna *NULL*. Următorul program, *hugeint.c*, utilizează funcția *halloc* pentru a aloca o matrice de 100000 octeți:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
  {
     long int i;
     int huge *matrice_mare;
     if ((matrice_mare = (int huge *) halloc (100000L,
           sizeof(long int))) == NULL)
       printf ("Eroare la alocarea matricei huge\n");
     else
```

```
    {
      printf("Completeaza matricea\n");
      for (i = 0; i < 100000L; i++)
      tablou_mare[i] = i % 32768;
      for (i = 0; i < 100000L; i++)
      printf ("%d ", tablou_mare[i]);
      hfree(matrice_mare);
    }
}
```

Observație: *De asemenea, așa cum ați învățat, limita dimensiunii matricei nu se aplică programelor pe care le creați în mediul Windows. De fapt, unele compilatoare Windows nu mai acceptă cuvântul cheie **huge** pentru declararea matricelor.*

MODIFICAREA DIMENSIUNILOR UNUI BLOC DE MEMORIE ALOCAT

C/C++ 601

Așa cum ați învățat, C permite programelor dumneavoastră să aloce memorie dinamică pe durata execuției. După ce alocați un bloc de memorie, va trebui probabil ca uneori să modificați dimensiunea blocului. În aceste cazuri, programele dumneavoastră pot utiliza funcția *realloc*, ca mai jos:

```
#include <stdlib.h>
void *realloc(void *bloc, size_t octeti_doriti);
```

Parametrul *bloc* este un pointer la memoria alocată anterior. Parametrul *octeti_doriti* este dimensiunea dorită pentru noul bloc. Funcția *realloc* poate să reducă sau să mărească blocul. Dacă funcția *realloc* reușește, ea va returna un pointer la bloc, care poate fi diferit de cel inițial. Cu alte cuvinte, funcția *realloc* poate muta blocul pentru a găsi spațiu (copiind datele, dacă este necesar). Dacă apare o eroare, funcția *realloc* returnează *NULL*. Următorul program, *realloc.c*, utilizează funcția *realloc* pentru mărirea dimensiunii blocului de la 100 de octeți la 1000 de octeți, astfel:

```
#include <stdio.h>
#include <alloc.h>
void main(void)
  {
    char *sir, *sir_nou;
    if ((sir = (char *) malloc(100)))
      {
        printf("Alocare reusita sir de 100 octeti\n");
        if ((sir_nou = (char *) realloc(sir, 1000)))
          printf("Dimensiunea sir marita la 1000\n");
        else
          printf("Eroare la realocarea sir\n");
      }
    else
      printf("Eroare la alocarea sir de 100 octeti\n");
  }
```

602 FUNCȚIA BRK



Așa cum ați învățat, zona *heap* începe la locația octetului care urmează imediat după ultimul octet din segmentul de date. O *valoare break* este adresa la care începe zona *heap*. Funcția *brk* permite programelor dumneavoastră să modifice valoarea *break*, atribuind-o unei anumite adrese, ca mai jos:

```
#include <alloc.h>

int brk(void *adresa);
```

Dacă funcția *brk* reușește, ea va returna valoarea 0. Dacă apare o eroare, funcția *brk* returnează –1. Următorul program, *brk.c*, utilizează funcția *brk* pentru a stabili valoarea break la 512 octeți înaintea locației curente. Programul utilizează funcția *coreleft* pentru a afișa volumul din zona *heap* disponibil înainte și după operația *brk*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *ptr;

    printf("Incepe heap disponibil %u\n", coreleft());
    ptr = malloc(1); // Reda un pointer la valoarea break curenta
    if (brk(ptr + 512) == 0)
    printf("Se incheie heap disponibil %u\n", coreleft());
  }
```

603 VALIDAREA ZONEI HEAP



Dacă ați întâlnit erori într-un program care alocă memorie dinamică și nu puteți să localizați sursa erorii, puteți să apelați la *validarea zonei heap*. Pentru a facilita testarea stării zonei *heap*, multe compilatoare dispun de o serie de rutine de bibliotecă run-time, cum ar fi *heapwalk* și *heapcheck*. Câteva dintre secțiunile care urmează prezintă modalități prin care programele dumneavoastră pot testa zona *heap*.

604 EXECUTAREA UNEI VERIFICĂRI RAPIDE A ZONEI HEAP



Așa cum ați învățat, pentru facilitarea localizării erorilor din programele dumneavoastră care execută o alocare de memorie dinamică, puteți să verificați starea zonei *heap*. Una dintre rutinele pe care programele dumneavoastră o pot utiliza pentru verificarea zonei *heap* este *heapcheck*:

```
#include <alloc.h>

int heapcheck(void);
```

Funcția *heapcheck* trece prin zona *heap* și examinează fiecare intrare din zonă. Funcția returnează una dintre valorile listate în tabelul 604.

Valoare	Descriere
_HEAPEMPTY	Nu există *heap*
_HEAPOK	*Heap* este verificat
_HEAPCORRUPT	Una sau mai multe intrări alterate

Tabelul 604 *Valorile returnate de funcția* ***heapcheck.***

Următorul program, *heapcbk.c*, utilizează funcția *heapcheck* pentru testarea stării zonei *heap*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *buffer, *al_doilea _buffer;
    int i, stare;

    buffer = malloc(100);
    al_doilea_buffer = malloc(100);
    stare = heapcheck();
    if (stare == _HEAPOK)
      printf("Heap este ok\n");
    else if (stare == _HEAPCORRUPT)
      printf("Heap este alterat\n");
    for (i = 0; i <= 100; i++)
      buffer[i] = i;
    stare = heapcheck();
    if (stare == _HEAPOK)
      printf("Heap este ok\n");
    else if (stare == _HEAPCORRUPT)
      printf("Heap este alterat\n");
  }
```

Atunci când programul alocă pentru prima dată memorie, funcția *heapcheck* returnează valoarea de stare, care este OK. După ce programul atribuie valori la buffer, funcția *heapcheck* returnează valoarea de stare ce spune că „*Heap* este alterat". Dacă examinați cu atenție bucla *for*, veți observa că ea atribuie 101 valori la bufferul de 100 de octeți (ceea ce alterează intrarea). Utilizând funcția *heapcheck*, puteți să detectați foarte rapid astfel de erori.

COMPLETAREA SPAȚIULUI LIBER AL ZONEI HEAP

C/C++ 605

O modalitate de detectare a erorilor de utilizare a memoriei în programele care lucrează cu memorie dinamică este completarea întregului spațiu liber al zonei *heap* cu o valoare specificată. Astfel, când executați operații cu memoria, puteți testa dacă valoarea s-a modificat. Pentru facilitarea completării și testării spațiului liber al zonei *heap*, multe din compilatoarele C dispun de următoarele funcții:

```
#include <alloc.h>
int heapcheckfree(unsigned int val);
int heapfillfree(unsigned int val);
```

Parametrul *valoare* este valoarea pe care vreţi să o atribuiţi în spaţiul liber al zonei *heap*. Funcţiile returnează una dintre valorile listate în tabelul 605.

Valoare	Descriere
_HEAPEMPTY	Nu există *heap*
_HEAPOK	Zona *heap* este verificată
_HEAPCORRUPT	Una sau mai multe intrări alterate
_BADVALUE	A fost întâlnită o altă valoare

Tabelul 605 *Valorile returnate de funcţiile* ***heapcheckfree*** *şi* ***heapfillfree.***

Următorul program, *compheap.c*, utilizează funcţiile *heapcheckfree* şi *heapfillfree*.

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    char *buffer1, *buffer2, *buffer3;
    int i, stare;

    buffer1 = malloc(100);
    buffer2 = malloc(200);
    buffer3 = malloc(300);
    free(buffer2); // Spatiu liber in al doilea
    stare = heapfillfree('A');
    if (stare == _HEAPOK)
      printf("Heap este ok\n");
    else if (stare == _HEAPCORRUPT)
      printf("Heap este alterat\n");
    for (i = 0; i <= 150; i++)
      buffer1[i] = i;
    stare = heapcheckfree('A');
    if (stare == _HEAPOK)
      printf("Heap este ok\n");
    else if (stare == _HEAPCORRUPT)
      printf("Heap este alterat\n");
    else if (stare == _BADVALUE)
      printf("Valoare modificata in spatiul liber\n");
  }
```

VERIFICAREA UNEI INTRĂRI SPECIFICATE A ZONEI HEAP

C/C++ 606

În secțiunea 604 ați învățat cum se utilizează funcția *heapcheck* pentru testarea stării întregii zone *heap*. Când verificați existența erorilor, puteți să testați starea individuală a intrărilor zonei *heap*. Pentru executarea acestui test, programele dumneavoastră pot utiliza funcția *heapchecknode*:

```
#include <alloc.h>
int heapchecknode(void *bloc);
```

Parametrul *bloc* este un pointer la blocul de memorie dinamică alocată. Funcția va returna una dintre valorile arătate în tabelul 606.

Valoare	Descriere
_HEAPEMPTY	Nu există *heap*
_HEAPOK	*Heap* este verificat
_HEAPCORRUPT	Una sau mai multe intrări alterate
_BADNODE	Blocul nu a fost găsit
_FREEENTRY	Blocul este liber
_USEDENTRY	Blocul este în uz

Tabelul 606 *Valorile returnate de funcția* ***heapchecknode.***

Următorul program, *heapnode.c*, ilustrează modul de utilizare a funcției *heapchecknode*:

```
#include <stdio.h>
#include <alloc.h>
void main(void)
  {
    char *buffer, *al_doilea_buffer;
    int i, stare;
    buffer = malloc(100);
    al_doilea_buffer = malloc(100);
    stare = heapchecknode(buffer);
    if (stare == _USEDENTRY)
      printf("buffer este ok\n");
    else
      printf("buffer nu este ok\n");
    stare = heapchecknode(al_doilea_buffer);
    if (stare == _USEDENTRY)
      printf("al_doilea_buffer este ok\n");
    else
      printf("al_doilea_buffer nu este ok\n");
    for (i = 0; i <= 100; i++)
      buffer[i] = i;
    stare = heapchecknode(buffer);
```

```
   if (stare == _USEDENTRY)
     printf("buffer este ok\n");
   else
     printf("buffer nu este ok\n");
   stare = heapchecknode(al_doilea_buffer);
   if (stare == _USEDENTRY)
     printf("al_doilea_buffer este ok\n");
   printf("al_doilea_buffer nu este ok\n");
 }
```

607 PARCURGEREA INTRĂRILOR ZONEI HEAP

C/C++

Pentru a vă ajuta să examinați individual intrările zonei *heap*, multe compilatoare de C dispun de o funcția numită *heapwalk*. Funcția *heapwalk* vă permite afișarea dimensiunii și stării (fie în uz, fie disponibilă) a fiecărei intrări din zona *heap*:

```
#include <alloc.h>

int heapwalk(struct heapinfo *info);
```

Parametrul *info* este un pointer la o structură de tip *heapinfo*:

```
struct heapinfo
   {
     void *pointer;
     unsigned int size;
     int in_use;
   };
```

Înainte de prima apelare a funcției *heapwalk*, trebuie să stabiliți membrul *pointer* al structurii *heapinfo* la valoarea *NULL*. Funcția *heapwalk* returnează una dintre valorile prezentate în tabelul 607.

Valoare	Descriere
_HEAPEMPTY	Nu există *heap*
_HEAPOK	*Heap* este verificat
_HEAPEND	Ultima intrare în *heap*

Tabelul 607 *Valorile returnate de funcția* ***heapwalk.***

Următorul program, *heapwalk.c*, parcurge intrările zonei *heap* utilizând funcția *heapwalk*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
   {
     char *buffer1, *buffer2, *buffer3;
     struct heapinfo node = { NULL, 0, 0};
     buffer1 = malloc(100);
```

```
    buffer2 = malloc(200);
    buffer3 = malloc(300);
    free(buffer2);
    while (heapwalk(&node) == _HEAPOK)
      printf("Dimensiune %u octeti Stare %s\n", node.size,
             (node.in_use) ? "In uz": "Liber");
  }
```

PRIVIND ÎNTR-O ANUMITĂ LOCAȚIE DE MEMORIE

C/C++ 608

În conformitate cu funcțiile programelor dumneavoastră, uneori poate va fi nevoie ca programul să acceseze o anumită locație de segment și deplasament din memorie. Dacă lucrați cu pointeri *far*, puteți să combinați un segment și un deplasament pentru a localiza adresa utilizând *MK_FP*. În plus, programele dumneavoastră pot utiliza funcțiile *peekb* și *peek*:

```
#include <dos.h>

char peekb(unsigned segment, unsigned deplasament);
int peek(unsigned segment, unsigned deplasament);
```

Parametrii *segment* și *deplasament* se combină pentru a specifica locația de memorie dorită. Următorul program, *ecr_fis.c*, utilizează funcția *peekb* pentru a capta conținutul curent al ecranului (în mod text) și pentru a trimite copia într-un fișier *cop_ecr.dat*. Programul privește (citește) octetul de caracter și de atribut. De aceea, programul *ecr_fis.c* trebuie să privească 4000 de caractere și 4000 de atribute:

```
#include <stdio.h>
#include <dos.h>

#define VIDEO 0xB800 // CGA base
void main(void)
  {
    FILE *pointer_fisier;
    int deplasament;
    if ((pointer_fisier = fopen("COP_ECR.DAT", "wb")) == NULL)
      printf("Eroare la deschiderea fisierului\n");
    else
      {
        for (deplasament = 0; deplasament < 8000; deplasament++)
        fprintf(pointer_fisier, "%c", peekb(VIDEO, deplasament));
        fclose(pointer_fisier);
      }
  }
```

Observație: *Programul* ***ecr_fis.c*** *utilizează adresa de bază a adaptorului video CGA d B800H. Dacă utilizați un adaptor video EGA, VGA sau altul, puteți fi obligat să schimbaț această adresă de bază.*

609 INTRODUCEREA DE VALORI ÎN MEMORIE



În secțiunea 608 ați învățat cum se utilizează funcțiile *peekb* și *peek* pentru a citi valori din adresele de segment și de deplasament de memorie specificate. În mod similar, majoritatea compilatoarelor C dispun de funcțiile *poke* și *pokeb*, care permit programelor dumneavoastră să plaseze valori la o locație de memorie specificată:

```
#include <dos.h>

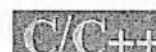
void pokeb(unsigned segment, unsigned deplasament, char valoare);
int poke(unsigned segment, unsigned deplasament, int valoare);
```

Următorul program, *ecr_poke.c*, utilizează funcția *pokeb* pentru a restabili conținutul ecranului pe care programul *ecr_fis.c* l-a salvat anterior:

```
#include <stdio.h>
#include <dos.h>

#define VIDEO 0xB800 // CGA base
void main(void)
   {
     FILE *pointer_fisier;
     int deplasament;
     char val;
     if ((pointer_fisier = fopen("COP_ECR.DAT", "rb")) == NULL)
       printf("Eroare la deschiderea fisierului\n");
     else
       {
         for (deplasament = 0; deplasament < 8000; deplasament++)
           {
             fscanf(pointer_fisier, "%c", &val);
             pokeb(VIDEO, deplasament, val);
           }
         fclose(pointer_fisier);
       }
   }
```

610 PORTURILE PC-ULUI



PC-ul utilizează două tehnici pentru comunicarea cu dispozitivele hardware interne. Conform primei tehnici, PC-ul poate să facă referință la locațiile de memorie pe care dispozitivul sau PC-ul le-a rezervat anterior pentru dispozitiv. Termenul pentru operațiile de intrare și ieșire care au loc prin astfel de locații de memorie este *I/O mapate în memorie*. PC-ul utilizează I/O mapate în memorie pentru efectuarea ieșirii video. În plus, PC-ul poate comunica cu dispozitivele hardware utilizând *porturile*. Cel mai bine vă imaginați un port ca un registru în care PC-ul sau dispozitivul poate plasa valori specifice. Tabelul 610 listează adresele de port pe care le utilizează diferite dispozitive într-un sistem EISA.

Port	Dispozitiv	Port	Dispozitiv
00H-1FH	Controler DMA	2EFH-2FFH	COM2
20H-3FH	Controler de întreruperi	300H-31FH	Plăci de rețea
40H-5FH	Cronometru de sistem	378H-37FH	LPT1
60H-6FH	Tastatură	380H-38FH	SDLC
70H-7FH	Ceas de timp real	390H-39FH	Adaptor unitate de alocare (cluster)
80H-9FH	Registre de pagină DMA	3B0H-3BFH	Monocrom
A0H-BFH	Controler 2 *int*	3C0H-3CFH	EGA
C0H-DFH	Controler 2 DMA	3D0H-3DFH	CGA
F0H-FFH	Coprocesor matematic	3F0H-3F7H	Dischetă
1F0H-1FFH	Hard-disc	3F8H-3FFH	COM1
200H-220H	Adaptor jocuri	400H-4FFH	DMA
270H-27FH	LPT2	500H-7FFH	Alias pentru 100H-3FFH
2B0H-2DFH	Substitut EGA	800H-8FFH	CMOS
2E0H-2E7H	COM4	900H-9FFH	Rezervat
2E8H-2EFH	COM3	9FFH-FFFFH	Rezervat

Tabelul 610 *Adresele de port ale PC-ului.*

Înțelesul fiecărui port depinde de dispozitivul corespunzător. Pentru a afla semnificațiile specifice ale porturilor, consultați documentația tehnică a PC-ului sau a dispozitivului.

ACCESUL LA VALORILE DE PORT

C/C++ 611

Dacă programele dumneavoastră execută controlul hardware de nivel inferior, puteți să citiți sau să scrieți o valoare de port. Pentru a ajuta programele să facă aceasta, majoritatea compilatoarelor C de mediu DOS dispun de următoarele funcții:

```
#include <dos.h>

int inport(int adresa_port);
char inportb(int adresa_port);
void outport(int adresa_port, int valoare);
void outportb(int adresa_port, unsigned char valoare);
```

Parametrul *adresa_port* specifică adresa portului dorit, așa cum sunt listate în tabelul 610. Parametrul *valoare* conține cuvântul sau valoarea de octet pe care programul dumneavoastră o va transmite la ieșire către port. Secțiunea 612 ilustrează modul de utilizare a funcției *inportb* pentru a citi și afișa conținutul memoriei CMOS a PC-ului.

CMOS

C/C++ 612

După cum știți, PC-ul stochează informații despre configurația sistemului în memoria CMOS, inclusiv tipurile de unitate, data sistemului și așa mai departe. PC-ul nu accesează CMOS utilizând modul de adresare standard cu segment și deplasament. În schimb, PC-ul utilizează

adresele de port PC pentru comunicarea cu CMOS. Aşa cum aţi învăţat în secţiunea 611, majoritatea compilatoarelor de C dispun de funcţii cum ar fi *inport* şi *outport* pentru a ajuta programele să acceseze porturile PC-ului. Următorul program, *infocmos.c*, utilizează funcţia *inportb* pentru a obţine şi afişa informaţii despre CMOS:

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
  {
    struct CMOS {
      unsigned char current_second;
      unsigned char alarm_second;
      unsigned char current_minute;
      unsigned char alarm_minute;
      unsigned char current_hour;
      unsigned char alarm_hour;
      unsigned char current_day_of_week;
      unsigned char current_day;
      unsigned char current_month;
      unsigned char current_year;
      unsigned char status_registers[4];
      unsigned char diagnostic_status;
      unsigned char shutdown_code;
      unsigned char drive_types;
      unsigned char reserved_x;
      unsigned char disk_1_type;
      unsigned char reserved;
      unsigned char equipment;
      unsigned char lo_mem_base;
      unsigned char hi_mem_base;
      unsigned char hi_exp_base;
      unsigned char lo_exp_base;
      unsigned char fdisk_0_type;
      unsigned char fdisk_1_type;
      unsigned char reserved_2[19];
      unsigned char hi_check_sum;
      unsigned char lo_check_sum;
      unsigned char lo_actual_exp;
      unsigned char hi_actual_exp;
      unsigned char century;
      unsigned char information;
      unsigned char reserved3[12];
    } cmos;
    char i;

    char *pointer;
    char octet;
```

```
   pointer = (char *) &cmos;
   for (i = 0; i < 0x34; i++)
     {
       outportb(0x70, i);
       octet = inportb(0x71);
       *pointer++ = octet;
     }
   printf("Data curenta %d/%d/%d\n", cmos.current_month,
          cmos.current_day, cmos.current_year);
   printf("Ora curenta %d:%d:%d\n", cmos.current_hour,
          cmos.current_minute, cmos.current_second);
   printf("Tipul de hard-disc %d\n", cmos.fdisk_0_type);
 }
```

MODELELE DE MEMORIE

C/C++ 613

Atunci când creați programe în mediu PC, compilatorul utilizează un *model de memorie* pentru a determina volumul de memorie pe care sistemul de operare îl alocă programului. Așa cum ați învățat, PC-ul împarte memoria în blocuri de 64Kb numite segmente. În mod obișnuit, programul dumneavoastră utilizează un segment pentru cod (instrucțiunile programului) și un al doilea segment pentru date. Dacă programul dumneavoastră este foarte mare sau utilizează un volum mare de date, probabil că uneori compilatorul va trebui să dispună de mai multe segmente de cod sau de date, sau de ambele. Modelul de memorie definește numărul de segmente pe care compilatorul le poate folosi pentru fiecare. Modelele de memorie sunt importante pentru că, dacă utilizați un model de memorie necorespunzător, programul poate să nu dețină suficientă memorie pentru execuție.

De obicei, compilatorul va selecta modelul de memorie care este suficient de mare pentru programul dumneavoastră. Însă, așa cum ați învățat, cu cât modelul de memorie este mai mare, cu atât programul dumneavoastră se va executa mai lent. De aceea, scopul dumneavoastră este să utilizați cel mai mic model de memorie care să satisfacă necesitățile programului. Majoritatea compilatoarelor acceptă modelele de memorie *tiny*, *small*, *medium*, *compact*, *large* și *huge*. Câteva dintre secțiunile care urmează descriu aceste modele de memorie în detaliu. Pentru selectarea unui anumit model de memorie, veți include, de regulă, o opțiune în cadrul liniei de comandă a compilatorului. Consultați documentația care însoțește compilatorul pentru a determina opțiunile de model de memorie.

Observație: *Așa cum ați învățat, diferitele tipuri de modele de memorie utilizate pentru scrierea programelor C/C++ în mediu DOS nu se aplică în mediul Windows, care utilizează numai modelul de memorie virtuală. Deși următoarele șapte secțiuni sunt utile în cazul în care intenționați să scrieți sau nu programe DOS, ele sunt pur informative pentru prograO matorii Windows.*

MODELUL DE MEMORIE TINY

C/C++ 614

Un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Cel mai mic și mai rapid model de memorie este modelul *tiny*. Datorită naturii sale compacte, modelul de memorie *tiny* consumă cel mai mic

volum de memorie și se încarcă mai repede decât oricare alt model. Așa cum arată figura 614, modelul de memorie *tiny* combină codul programului și datele într-un singur segment de 64Kb.

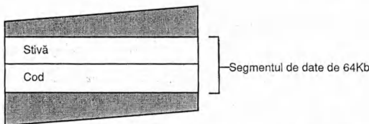


Figura 614 *Modelul de memorie tiny plasează codul programului și datele într-un singur segment de 64Kb.*

Când creați programe de mici dimensiuni, așa cum sunt multe dintre exemplele de programe prezentate pe parcursul acestei cărți, puteți cere compilatorului să utilizeze modelul de memorie *tiny*.

615 *MODELUL DE MEMORIE SMALL*



Un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Cel mai obișnuit model de memorie este modelul *small*. Așa cum arată figura 615, modelul de memorie *small* utilizează un segment de 64Kb pentru codul programului și un al doilea pentru datele programului.

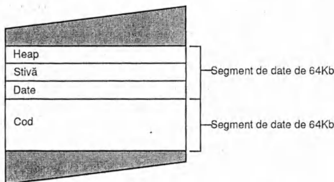


Figura 615 *Modelul de memorie small utilizează un segment pentru codul programului și un altul pentru date.*

Avantajul utilizării modelului de memorie *small* este acela că toate apelările de funcții și toate referințele la date utilizează adrese *near* de 16 biți. Astfel, un program se va executa mai repede decât cele care utilizează alte modele, mai mari de memorie.

MODELUL DE MEMORIE MEDIUM

C/C++616

Un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Dacă programul dumneavoastră necesită mai mult decât 64Kb de memorie pentru cod, dar numai 64Kb (sau mai puțin) pentru date, atunci programul poate utiliza modelul de memorie *medium*. Așa cum arată figura 616, modelul de memorie *medium* alocă mai multe segmente de cod și numai un segment de date.

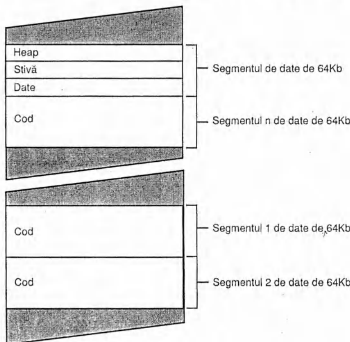


Figura 616 *Modelul de memorie medium alocă mai multe segmente de cod și un segment de date.*

Dacă programul dumneavoastră conține un număr mare de instrucțiuni, atunci modelul de memorie *medium* favorizează accesul rapid la date datorită faptului că toate referințele la date utilizează adrese *near*. Pentru că modelul de memorie *medium* utilizează mai multe segmente cod, totuși, toate apelurile de funcții cer adrese *far* pe 32 biți. Depunerea și extragerea (în și din stivă) a adreselor suplimentare de segmente pentru apelurile de funcții vor scădea ușor performanța programului.

MODELUL DE MEMORIE COMPACT

C/C++617

Așa cum ați învățat, un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Dacă programul dumneavoastră

utilizează un volum mare de date, dar instrucțiuni limitate, programul poate să utilizeze modelul de memorie *compact*. Așa cum arată figura 617, modelul de memorie *compact* alocă un segment de 64Kb pentru codul programului dumneavoastră și mai multe segmente pentru date.

Date — Segmentul *n* de date de 64Kb

Date — Segmentul 2 de date de 64Kb

Heap
Stivă — Segmentul 1 de date de 64Kb
Date

Cod — Segmentul de date de 64Kb

***Figura 617** Modelul de memorie compact alocă un segment de 64Kb pentru cod și mai multe segmente pentru date.*

Datorită faptului că modelul de memorie *compact* utilizează un segment cod, toate apelările de funcții utilizează adrese *near* de 16Kb. Ca urmare, apelurile funcțiilor vor fi mai rapide decât în oricare model de memorie mai mare. Referințele la date, pe de altă parte, cer o adresă de segment și deplasament (adresă *far* de 32 de biți). Suprasarcina solicitată pentru lucrul cu fiecare adresă de segment și deplasament a referințelor la date va reduce performanța programului dumneavoastră.

618 MODELUL DE MEMORIE LARGE



Un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Dacă programul conține un volum mare de cod și date, programul poate utiliza modelul de memorie *large*. Așa cum arată figura 618, modelul de memorie large alocă mai multe segmente pentru cod și pentru date.

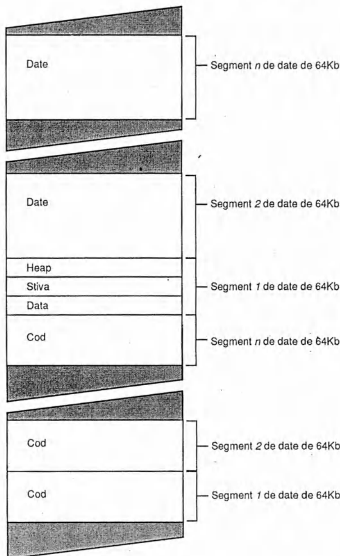


Figura 618 *Modelul de memorie large alocă mai multe segmente pentru cod și date.*

Ar trebui să utilizați modelul de memorie *large* numai ca ultimă resursă. Datorită faptului că modelul de memorie *large* utilizează mai multe segmente pentru cod și date, fiecare apelare de funcție și fiecare referință la date cere o adresă *far* pe 32 de biți. Suprasarcina asociată cu manipularea constantă de segmente și deplasamente face ca modelul de memorie *large* să fie cel mai lent dintre modelele descrise până acum.

619 MODELUL DE MEMORIE HUGE



Un model de memorie descrie numărul de segmente de memorie de 64Kb pe care compilatorul le alocă pentru un program. Aşa cum aţi învăţat, majoritatea compilatoarelor C pentru PC-uri dispun de mai multe modele diferite de memorie pentru a satisface solicitările datelor şi codului programelor dumneavoastră. O condiţie specială apare, însă, când programele dumneavoastră utilizează o matrice mai mare de 64Kb. Pentru a aloca o astfel de matrice, programul trebuie să utilizeze cuvântul cheie *huge* pentru a crea un pointer, ca mai jos:

```
int huge *matrice_mare;
```

Apoi, programul trebuie să utilizeze funcţia *halloc* pentru alocarea memoriei. Următorul program, *hugeint.c*, prezentat iniţial în secţiunea 600, utilizează funcţia *halloc* pentru a aloca o matrice de 400000 octeţi:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
  {
     long int i;
     int huge *matrice_mare;
     if ((matrice_mare = (int huge *) halloc (100000L,
       sizeof(long int))) == NULL)
      printf ("Eroare la alocarea matricei huge\n");
     else
      {
        printf("Completeaza matricea\n");
        for (i = 0; i < 100000L; i++)
           matrice_mare[i] = i % 32768;
        for (i = 0; i < 100000L; i++)
         printf ("%d ", matrice_mare[i]);
        hfree(matrice_mare);
      }
  }
```

Atunci când compilaţi şi executaţi programul utilizând modelul de memorie *huge*, majoritatea compilatoarelor vor utiliza adrese *far* pe 32 biţi atât pentru cod, cât şi pentru date (în mod similar cu modelul de memorie *large*). Prin urmare, execuţia programului poate fi mai lentă decât doriţi.

620 DETERMINAREA MODELULUI CURENT DE MEMORIE



În funcţie de prelucrarea programului, probabil că uneori va trebui să compilaţi programul utilizând un anumit model de memorie. Majoritatea compilatoarelor predefinesc o constantă specifică pentru a ajuta programele să determine modelul curent de memorie. Tabelul 620, de exemplu, enumeră constantele definite de compilatoarele *Turbo C++ Lite, Microsoft C* şi *Borland C* pentru diferitele modele de memorie

Model de memorie	Constante Microsoft C	Constante Turbo C++ Lite/Borland C
Small	*M_I86SM*	*_SMALL_*
Medium	*M_I86MM*	*_MEDIUM_*
Compact	*M_I86CM*	*_COMPACT_*
Large	*M_I86LM*	*_LARGE_*

***Tabelul 620** Constantele pe care le definesc Turbo C++ Lite, Microsoft C++ și Borland C pentru indicarea modelului curent de memorie.*

Dacă programul dumneavoastră solicită un anumit model de memorie, programul poate verifica modelul, ca mai jos:

```
#ifndef _MEDIUM_
  printf("Programul cere modelul de memorie medium\n");
  exit(1);
#endif
```

DATA ȘI ORA CURENTE CA SECUNDE DE LA 1/1/1970

C/C++ 621

Pe măsură ce programele dumneavoastră devin tot mai funcționale, ele vor avea nevoie adesea să cunoască data și ora curente. Majoritatea compilatoarelor de C dispun de câteva funcții care returnează data și ora în diferite formate. O asemenea funcție este *time*, care returnează data și ora curente ca secunde de la 00:00, 1 ianuarie 1970. Funcția returnează o valoare de tip *time_t*, ca mai jos:

```
#include <time.h>
time_t time(time_t *data_ora);
```

Dacă nu vreți să transmiteți un parametru funcției *time*, puteți să invocați funcția cu *NULL*:

```
timp_curent = time(NULL);
```

Următorul program, *pauza_5.c*, utilizează funcția *time* pentru a introduce o întârziere de 5 secunde:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
   time_t timp_curent;
   time_t timp_start;
   printf("Pe punctul de a face o intarziere de 5 secunde\n");
   time(&timp_start); // Timpul in secunde la start
   do {
     time(&timp_curent);
   }
   while ((timp_curent - timp_start) < 5);
```

```
    printf("Gata\n");
}
```

622 CONVERSIA DATEI ȘI OREI DIN SECUNDE ÎN ASCII

C/C++

Secțiunea 621 a prezentat funcția *time* care returnează ora și data curente ca secunde de la 00:00, 1 ianuarie 1970. Utilizând funcția *ctime*, programele dumneavoastră pot converti secundele în șir de caractere de următorul format:

```
"Fri Oct 31 11:30:00 1997\n"
```

Următorul program, *ctime.c*, ilustrează modul de utilizare a funcției *ctime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
{
    time_t timp_curent;
    time(&timp_curent); // Timpul in secunde;
    printf("Data si ora curente: %s", ctime(&timp_curent));
}
```

623 TRECEREA AUTOMATĂ LA ORARUL DE VARĂ

C/C++

Câteva dintre funcțiile prezentate de această secțiune țin seama de orarul de vară. Pentru a executa o astfel de prelucrare, multe compilatoare de C declară o variabilă globală numită *daylight*. Dacă orarul de vară este în vigoare, variabila conține valoarea 1. Dacă nu este în vigoare, compilatorul de C stabilește variabila la 0. Funcțiile *tzset*, *localtime* și *ftime* controlează valoarea variabilei. Următorul fragment de cod utilizează variabila *daylight* pentru a determina dacă este în vigoare orarul de vară sau orarul standard:

```
if (daylight)
   printf("Orarul de vara este in vigoare\n");
else
   printf("Nu este in vigoare orarul de vara\n");
```

Observație: *Funcția* ***tzset*** *atribuie valoarea variabilei* ***daylight***.

624 ÎNTÂRZIEREA CU UN ANUMIT NUMĂR DE MILISECUNDE

C/C++

În funcție de programul dumneavoastră, poate că uneori va fi nevoie ca programul să aștepte un anumit număr de milisecunde (1/1000 secunde). De exemplu, puteți să afișați un mesaj pe ecran pentru câteva secunde, după care programul să continue fără ca utilizatorul să fie obligat să apese o tastă. Pentru asemenea cazuri, multe compilatoare de C dispun de funcția *delay*. Funcția va face o întârziere de un anumit număr de milisecunde:

```
#include <dos.h>
void delay(unsigned milisecunde);
```

Utilizând funcția *delay*, programele dumneavoastră pot specifica o suspendare de până la 65535 milisecunde. Următorul program, *delay.c*, utilizează funcția *delay* pentru o întârziere de cinci secunde:

```
#include <stdio.h>
#include <dos.h>

void main(void)
   {
     printf("Pe punctul de a face o intarziere de 5 secunde\n");
     delay(5000);
     printf("Gata\n");
   }
```

DETERMINAREA TIMPULUI DE PROCESARE AL PROGRAMULUI

C/C++625

Pe măsură ce avansați în creșterea performanței programului dumneavoastră, puteți să măsurați durata diferitelor părți ale programului. Puteți determina care dintre secțiunile programului sunt mari consumatoare de timp. De regulă, trebuie să începeți optimizarea de la secțiunea programului care consumă cel mai mare timp de procesare. Pentru a vă ajuta să determinați timpul de procesare al programelor dumneavoastră, compilatorul de C dispune de funcția *clock* care returnează numărul de unități de ceas (care de obicei apar de 18,2 ori pe secundă):

```
#include <time.h>
clock_t clock(void);
```

Funcția *clock* returnează timpul de procesare al programului în unități de ceas. Pentru a converti timpul în secunde, împărțiți rezultatul la constanta *CLK_TCK* care este definită în fișierul antet *time.h*. Următorul program, *clock.c*, utilizează funcția *clock* pentru a afișa timpul de procesare al programului în secunde:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>     // Contine prototipul functiei delay

void main(void)
   {
     clock_t timp_procesare;
     printf("Timp de procesare consumat %ld\n", (long) clock() /
        (long) CLK_TCK);
     delay(2000);
     printf("Timp de procesare consumat %ld\n", (long) clock() /
        (long) CLK_TCK);
     delay(3000);
```

```
    printf("Timp de procesare consumat %ld\n", clock() /
      (long) CLK_TCK);
  }
```

Observație: *Când compilatorul nu oferă funcția delay, puteți folosi una sau mai multe bucle for pentru a implementa o întârziere.*

626 COMPARAREA ÎNTRE DOUĂ VALORI DE TIMP

C/C++

În secțiunea 621 ați învățat cum se utilizează funcția *time* pentru a obține numărul de secunde de la 1 ianuarie 1970. Atunci când lucrați cu măsurarea timpului, programele dumneavoastră vor trebui adeseori să compare două sau mai multe valori de timp. Pentru a compara valori de timp, programul dumneavoastră poate utiliza funcția limbajului C *difftime*, care returnează diferența dintre două momente, ca o valoare în virgulă mobilă, ca mai jos:

```
float difftime(time_t timp_final, time_t timp_start);
```

Următorul program, *difftime.c*, utilizează funcția *difftime* pentru o întârziere de 5 secunde:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
    time_t timp_start;
    time_t timp_curent;
    time(&timp_start);
    printf("Pe punctul de a face o intarziere de 5 secunde\n");
    do
      {
        time(&timp_curent);
      } while (difftime(timp_curent, timp_start) < 5.0);
    printf("Gata\n");
  }
```

627 OBȚINEREA DATEI SUB FORMA UNUI ȘIR DE CARACTERE

C/C++

În secțiunea 622 ați învățat cum se utilizează funcția *ctime* pentru a crea un șir de caractere care conține data și ora. Pentru utilizarea funcției *ctime*, trebuie ca mai întâi să invocați funcția *time* pentru a obține numărul de secunde de la 1 ianuarie 1970. Dacă doriți să obțineți numai data curentă, programele dumneavoastră pot utiliza funcția *_strdate*:

```
#include <dos.h>

char *_strdate(char *buffer_data);
```

Bufferul șir de caractere care este transmis funcției *_strdate* trebuie să fie suficient de mare pentru a putea stoca nouă caractere (opt caractere pentru dată și unul pentru *NULL*). Funcția *_strdate* redă data sub forma *mm/dd/yy* (luna/ziua/anul). Următorul program, *strdate.c*, utilizează funcția *strdate* pentru a afișa data curentă:

```
#include <stdio.h>
#include <time.h>

void main(void)
   {
     char data[9];
     _strdate(data);
     printf("Data curenta este %s\n", data);
   }
```

OBȚINEREA OREI SUB FORMA UNUI ȘIR DE CARACTERE

C/C++ 628

În secțiunea 622 ați învățat cum se utilizează funcția *ctime* pentru a crea un șir de caractere care conține data și ora. Pentru utilizarea funcției *ctime*, trebuie să invocați mai întâi funcția *time*, pentru a obține numărul de secunde de la 1 ianuarie 1970. Dacă doriți să obțineți numai ora curentă, programele dumneavoastră pot utiliza funcția *_strtime*, ca mai jos:

```
#include <dos.h>

char *_strtime(char *buffer_timp);
```

Bufferul șir de caractere care este transmis funcției *_strtime* trebuie să fie suficient de mare pentru a putea stoca nouă caractere (opt caractere pentru oră și unul pentru *NULL*). Funcția *_strtime* redă ora sub forma *hh/mm/ss* (ora/minutul/secunda). Următorul program, *strtime.c*, utilizează funcția *strtime* pentru a afișa ora curentă:

```
#include <stdio.h>
#include <time.h>

void main(void)
   {
     char ora[9];
     _strdate(ora);
     printf("Ora curenta este %s\n", ora);
   }
```

CITIREA CRONOMETRULUI BIOS

C/C++ 629

BIOS are încorporat un ceas intern care bate de 18,2 ori pe secundă. BIOS păstrează în cadrul memoriei numărul de unități de ceas care se scurg de la miezul nopții. În trecut, multe programe utilizau cronometrul BIOS pentru a întârzia programele până trecea un anumit număr de unități de ceas. Așa cum am explicat anterior, programele dumneavoastră pot specifica, însă, mult mai precis un interval de timp (la nivel de milisecunde) utilizând funcția *delay*. Cronometrul BIOS rămâne util pentru a genera punctul inițial al unui generator de numere aleatoare. Multe compilatoare de C dispun de două funcții care vă permit controlul asupra cronometrului BIOS – *biostime* și *_bios_timeofday*. Funcția *biostime* permite programelor să acceseze numărul de unități de ceas care au trecut de la miezul nopții. Formatul funcției *biostime* este următorul:

```
#include <bios.h>
long biostime(int operatie, long timpnou);
```

Parametrul *operatie* vă permite să specificați dacă vreți să citiți sau să fixați ceasul intern BIOS, cum arătăm în tabelul 629.1.

Valoare	Semnificație
0	Citește valoarea curentă a cronometrului
1	Fixează valoarea cronometrului la valoarea *timpnou*

Tabelul 629.1 *Valorile posibile ale parametrului* ***operatie.***

Funcția returnează numărul curent de unități de ceas. Funcția *_bios_timeofday* permite, de asemenea, citirea sau fixarea cronometrului BIOS:

```
#include <bios.h>
long _bios_timeofday(int operatie, long *batai);
```

Parametrul *operatie* vă permite și de această dată să specificați dacă vreți să citiți sau să fixați cronometrul, cum se arată în tabelul 629.2.

Valoare	Semnificație
_TIME_GETCLOCK	Citește valoarea curentă a cronometrului
_TIME_SETCLOCK	Fixează valoarea cronometrului la valoarea *din bătăi*

Tabelul 629.2 *Valorile posibile ale parametrului* ***operatie.***

Funcția *_bios_timeofday* returnează valoarea pe care serviciul de cronometru din BIOS o păstrează în registrul AX. Următorul program, *bioscron.c*, utilizează ambele funcții pentru a citi unitățile de ceas curente din BIOS:

```
#include <stdio.h>
#include <bios.h>
void main(void)
   {
    long batai;
    batai = biostime(0, batai);
    printf("Batai de la miezul noptii %ld\n", batai);
     _bios_timeofday(_TIME_GETCLOCK, &batai);
    printf("Secunde de la miezul noptii %f\n", batai / 18.2);
   }
```

630 LUCRUL CU ORA LOCALĂ



În secțiunea 621 ați învățat că funcția *time* returnează timpul curent în secunde de la miezul nopții, 1 ianuarie 1970. Pentru a face ca ora sistemului să fie mai ușor de folosit pentru programele dumneavoastră, compilatorul de C dispune de funcția *localtime*, care convertește timpul în secunde la o structură de tip *tm*. Structura *tm* este definită în fișierul antet *time.h* și este prezentată mai jos:

```
struct tm
   {
      int tm_sec;    // secunde de la 0 la 59
      int tm_min;    // minute de la 0 la 59
      int tm_hour;   // ore de la 0 la 24
      int tm_mday;   // ziua de la 1 la 31
      int tm_mon;    // luna de la 0 la 11
      int tm_year;   // anul - 1900
      int tm_wday;   // de la 0 pentru duminica la 6
                     // pentru sambata
      int tm_yday;   // ziua din an de la 1 la 365
      int tm_isdst;  // diferit de zero daca este in vigoare
                     //  orarul de vara
   };
```

Formatul funcției *localtime* este următorul:

```
#include <time.h>
struct tm *localtime(const time_t *cronometru);
```

Funcția *localtime* utilizează variabilele globale *timezone* și *daylight* pentru a potrivi ora la fusul orar al zonei dumneavoastră și pentru a ține seama de orarul de vară. Următorul program, *localtim.c*, ilustrează modul de utilizare a funcției *localtime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
   {
     struct tm *data_curenta;
     time_t secunde;
     time(&secunde);
     data_curenta = localtime(&secunde);
     printf("Data curenta: %d-%d-%d\n", data_curenta->tm_mon+1,
            data_curenta->tm_mday, data_curenta->tm_year);
     printf("Ora curenta: %02d:%02d\n", data_curenta->tm_hour,
            data_curenta->tm_min);
   }
```

LUCRUL CU ORA MERIDIANULUI GREENWICH C/C++ 631

În secțiunea 621 ați învățat că funcția *time* returnează timpul curent în secunde de la miezul nopții, 1 ianuarie 1970. Dacă lucrați cu utilizatori internaționali, probabil că uneori va trebui să folosiți ora de referință a meridianului Greenwich (GMT). Pentru ca să puteți lucra cu ora meridianului Greenwich, compilatorul de C dispune de funcția *gmtime* care convertește timpul în secunde la o structură de tip *tm*, prezentată în secțiunea 630. Formatul funcției *gmtime* este următorul:

```
#include <time.h>
struct tm *gmtime(const time_t *cronometru);
```

Funcția *gmtime* utilizează variabila globală *daylight* pentru a ține seama de orarul de vară. Următorul program, *gmtime.c*, ilustrează modul de utilizare a funcției *gmtime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
    struct tm * data_gm;
    time_t secunde;
    time(&secunde);
    data_gm = gmtime(&secunde);
    printf("Data curenta: %d-%d-%d\n", data_gm->tm_mon+1,
           data_gm->tm_mday, data_gm->tm_year);
    printf("Ora curenta: %02d:%02d\n", data_gm->tm_hour,
           data_gm->tm_min);
  }
```

632 OBȚINEREA TIMPULUI SISTEMULUI DOS



Dacă utilizați sistemul DOS, programele dumneavoastră pot utiliza funcția *gettime* în scopul de a obține ora de sistem DOS. Funcția *gettime* atribuie ora curentă la o structură de tip *time*, care este definită în fișierul antet *dos.h* și este arătată mai jos:

```
struct time
  {
    unsigned char ti_min; // minute de la 0 la 59
    unsigned char ti_hour; // ore de la 0 la 24
    unsigned char ti_hund; // sute de secunde de la 0 la 99
    unsigned char ti_sec; // secunde de la 0 la 59
  };
```

Formatul funcției *gettime* este următorul:

```
#include <dos.h>
void gettime(struct time *timp_curent);
```

Următorul program, *dostime.c*, utilizează funcția *gettime* pentru a obține și apoi a afișa ora curentă a sistemului:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct time timp_curent;
```

```
    gettime(&timp_curent);
    printf("Timpul curent %02d:%02d:%02d.%d\n",
      timp_curent.ti_hour, timp_curent.ti_min,
      timp_curent.ti_sec, timp_curent.ti_hund);
  }
```

Observație: *Multe compilatoare de C de mediu DOS dispun, de asemenea, de funcția* ***_dos_gettime*** *care returnează o structură de tip* ***dostime_t****, ca mai jos:*

```
struct dostime_t
  {
    unsigned char hour;       // ore de la 0 la 23
    unsigned char minute;     // minute de la 0 la 59
    unsigned char second;     // secunde de la 0 la 59
    unsigned char hsecond;    // secunde de la 0 la 99
  };
```

Formatul funcției *_dos_gettime* este următorul:

```
#include <dos.h>
void _dos_gettime(struct dostime_t *timp_curent);
```

Observație: *Compact discul care însoțește această carte include programul* ***wintime.cpp*** *care utilizează interfața API Windows pentru a obține ora curentă a sistemului.*

OBȚINEREA DATEI SISTEMULUI DOS

C/C++633

Dacă utilizați sistemul DOS, programele dumneavoastră pot utiliza funcția *getdate* pentru a obține data sistemului din DOS. Funcția *getdate* atribuie data curentă la o structură de tip *date*, care este definită în fișierul antet *dos.h* și este arătată mai jos:

```
struct date
  {
    int da_year; // anul curent
    char da_day; // ziua curenta de la 1 la 31
    char da_mon; // luna de la 1 la 12
  };
```

Formatul funcției *getdate* este următorul:

```
#include <dos.h>
void getdate(struct date *data_curenta);
```

Următorul program, *dosdata.c*, utilizează funcția *getdate* pentru a obține și apoi a afișa data curentă a sistemului:

```
#include <stdio.h>
#include <dos.h>
void main(void)
  {
```

```
    struct date data_curenta;
    getdate(&data_curenta);
    printf("Data curenta %d-%d-%d\n", data_curenta.da_mon,
           data_curenta.da_day, data_curenta.da_year);
  }
```

Observaţie: *Multe compilatoare de C de mediu DOS dispun, de asemenea, de funcţia **_dos_getdate** care returnează o structură de tip **dosdate_t**, ca mai jos:*

```
struct dosdate_t
  {
    unsigned char day;        // de la 1 la 31
    unsigned char month;      // de la 1 la 12
    unsigned int year;        // 1980-2099
    unsigned char dayofweek;  // de la 0 pentru duminica la
                              // 6 pentru sambata
  };
```

Formatul funcţiei *_dos_getdate* este următorul:

```
#include <dos.h>
void _dos_getdate(struct dosdate_t *data_curenta);
```

Observaţie: *Compact discul care însoţeşte această carte include programul **windate.cpp** care utilizează interfaţa API Windows pentru a obţine data curentă a sistemului.*

634 FIXAREA OREI SISTEMULUI DOS

C/C++

Dacă utilizaţi sistemul DOS, programele dumneavoastră pot utiliza funcţia *settime* pentru a fixa ora sistemului DOS exact cum ar reieşi din comanda DOS TIME. Pentru a utiliza funcţia *settime*, atribuiţi ora dorită la o structură de tip *time*, cum am arătat în secţiunea 632. Formatul funcţiei *settime* este următorul:

```
#include <dos.h>
void settime(struct time *timp_curent);
```

Următorul program, *settime.c*, utilizează funcţia *settime* pentru a fixa ora curentă a sistemului la 12:30:

```
#include <stdio.h>
#include <dos.h>
void main(void)
  {
    struct time ora_dorita;
    ora_dorita.ti_hour = 12;
    ora_dorita.ti_min = 30;
    settime(&ora_dorita);
  }
```

***Observaţie:** Multe compilatoare de C de mediu DOS dispun, de asemenea, de funcţia **_dos_settime** care foloseşte o structură de tip **dostime_t** pentru a fixa ora sistemului, cum se arată în secţiunea 632. Formatul funcţiei **_dos_settime** este următorul:*

```
#include <dos.h>

void _dos_settime(struct dostime_t *ora_curenta);
```

***Observaţie:** Compact discul care însoţeşte această carte include programul **wsettim.cpp** care utilizează interfaţa API Windows pentru a stabilirea ora curentă a sistemului.*

FIXAREA DATEI SISTEMULUI DOS

C/C++ 635

Dacă utilizaţi sistemul DOS, programele dumneavoastră pot utiliza funcţia *setdate* pentru a fixa data sistemului DOS. Înainte de a invoca funcţia *setdate*, atribuiţi data pe care o doriţi la o structură de tip *date*, cum am arătat în secţiunea 633. Apoi, utilizaţi un pointer la structură pentru a invoca funcţia. Formatul funcţiei *setdate* este următorul:

```
#include <dos.h>

void setdate(struct date *data_curenta);
```

Următorul program, *setdate.c*, utilizează funcţia *setdate* pentru a fixa data curentă a sistemului la 31 octombrie 1997:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    struct date data_dorita;
    data_dorita.da_mon = 10;
    data_dorita.da_day = 31;
    data_dorita.da_year = 1997;
    setdate(&data_dorita);
  }
```

***Observaţie:** Multe compilatoare de C de mediu DOS dispun, de asemenea, de funcţia **_dos_setdate** care utilizează o structură de tip **dosdate_t** pentru a fixa data sistemului, cum am arătat în secţiunea 633. Formatul funcţiei **_dos_setdate** este următorul:*

```
#include <dos.h>

void _dos_setdate(struct dosdate_t *data);
```

***Observaţie:** Compact discul care însoţeşte această carte include programul **winsdat.cpp** care utilizează interfaţa Windows API pentru a stabili data curentă a sistemului.*

CONVERSIA DATEI DOS ÎN FORMAT UNIX

C/C++ 636

În secţiunea 633 aţi învăţat cum se utilizează funcţia *getdate* pentru a obţine data sistemului DOS. De asemenea, în secţiunea 632 aţi învăţat cum se utilizează funcţia *gettime* pentru a obţine ora sistemului DOS. Dacă lucraţi într-un mediu în care utilizaţi şi DOS şi Unix, atunci

se poate întâmpla să fie nevoie să convertiți formatul datei și orei din sistem DOS la formatul datei și orei utilizat de Unix. În asemenea cazuri, programele dumneavoastră pot utiliza funcția *dostounix* pentru a efectua conversia. Funcția *dostounix* convertește structuri de tip *date* și de tip *time* la secunde de la miezul nopții, 1 ianuarie 1970:

```
#include <dos.h>
void dostounix(struct date *data_DOS, struct time *timp_DOS);
```

Următorul program, *dosunix.c*, utilizează funcția *dostounix* pentru a converti data și ora sistemului DOS la formatul corespunzător în Unix:

```
#include <stdio.h>
#include <dos.h>
#include <time.h>

void main(void)
  {
    struct time timpdos;
    struct date datados;
    time_t format_unix;
    struct tm *local;
    getdate(&datados);
    gettime(&timpdos);
    format_unix = dostounix(&datados, &timpdos);
    local = localtime(&format_unix);
    printf("Timp UNIX: %s\n", asctime(local));
  }
```

637 UTILIZAREA FUSELOR ORARE PENTRU A CALCULA DIFERENȚELE DE ORĂ

C/C++

Așa cum ați învățat, biblioteca run-time de C dispune de câteva funcții care pot converti valorile de oră între cea locală și cea a meridianului Greenwich. Pentru a ajuta programele dumneavoastră să determine rapid diferența de oră dintre două fuse orare, multe compilatoare de C dispun de funcția *timezone*, care conține numărul de secunde dintre două valori de timp. Următorul program, *timezone.c*, utilizează variabila globală *timezone* pentru a afișa diferența de oră:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
    tzset();
    printf("Diferenta intre ora locala si GMT este de %d
      ore\n", timezone / 3600);
  }
```

Observație: *Funcția **tzset** utilizează intrarea de mediu TZ pentru a determina fusul orar curent.*

DETERMINAREA FUSULUI ORAR CURENT

C/C++ 638

Câteva secțiuni din acest capitol au prezentat funcții care calculează ora pe baza fusului orar curent. Pentru a ajuta programele dumneavoastră să determine fusul orar curent, multe compilatoare de C dispun de variabila globală *tzname*. Variabila globală *tzname* conține doi pointeri: *tzname[0]* indică numele de fus orar cu trei caractere, iar *tzname[1]* indică numele zonei de orar de vară de trei caractere. Următorul program, *tzname.c*, utilizează variabila globală *tzname* pentru a afișa numele fuselor orare curente:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
    tzset();
    printf("Fusul orar curent este %s\n", tzname[0]);
    if (tzname[1])
      printf("Zona de orar de vara este %s\n", tzname[1]);
    else
      printf("Zona de orar de vara nu este definita\n");
  }
```

Observație: *Funcția* ***tzset*** *utilizează intrarea de mediu TZ pentru a determina fusul orar curent.*

FIXAREA FUSELOR ORARE CU FUNCȚIA TZSET

C/C++ 639

Câteva dintre funcțiile și variabilele globale din această secțiune returnează informații despre fusul orar curent. Multe funcții apelează funcția *tzset* pentru a obține informații despre fusul orar, ca mai jos:

```
#include <time.h>

void tzset(void);
```

Funcția *tzset* utilizează intrarea de mediu TZ pentru a determina valorile fusului orar. Funcția atribuie apoi valori corespunzătoare variabilelor globale *timezone, daylight* și *tzname*. Programul *tzname.c*, prezentat în secțiunea 638, ilustrează modul de utilizare a funcției *tzset*.

UTILIZAREA INTRĂRII DE MEDIU TZ

C/C++ 640

Multe dintre secțiunile prezentate de-a lungul acestui capitol apelează la funcția *tzset* pentru a obține informații despre fusul orar. Funcția *tzset* examinează intrările de mediu pentru intrarea TZ și atribuie apoi variabilele *timezone, daylight* și *tzname*, pe baza valorii intrării. Puteți utiliza fie comanda DOS SET pentru a atribui o valoare pentru intrarea TZ, fie stabilirea lui Windows Date/Time din Control Panel. Atunci când utilizați comanda DOS SET, formatul intrării este următorul:

```
TZ=SSS[+/-]h[h][DDD]
```

Unde SSS conține numele fusului orar standard (de exemplu, EST sau PST), *[+/-]h[h]* specifică diferența în ore între fusul orar standard și GMT; iar DDD specifică numele zonei de orar de vară (de exemplu, PDT). Următoarea intrare stabilește fusul orar pentru coasta de vest când este în vigoare orarul de vară:

```
C:\> SET TZ=PST8PDT <ENTER>
```

Se omite numele zonei de orar de vară atunci când orarul de vară nu este în vigoare, ca mai jos:

```
C:\> SET TZ=PST8 <ENTER>
```

Experimentați cu intrarea de mediu TZ și programele cu fusul orar prezentate în acest capitol pentru a determina cum vreți să reprezentați datele calendaristice și ora în cadrul programelor dumneavoastră. Amintiți-vă, însă, că programele dumneavoastră vor avea nevoie să scrie intrarea TZ pe orice calculator pe care vă mutați aplicațiile.

Observație: *Dacă nu specificați o intrare TZ, în mod implicit se stabilește EST5EDT.*

641 FIXAREA INTRĂRII DE MEDIU TZ DIN CADRUL UNUI PROGRAM



Așa cum ați învățat, câteva dintre funcțiile bibliotecii run-time de C utilizează funcția *tzset* pentru a determina fusul orar local. Cum am arătat în secțiunea 640, funcția *tzset* utilizează intrarea de mediu TZ pentru a determina fusul orar. În cele mai multe cazuri, nu este rezonabil să așteptați ca utilizatorii să stabilească corect intrarea de mediu TZ. Dacă, însă, cunoașteți valorile corecte pentru un anumit utilizator, puteți utiliza funcția *putenv* în cadrul programelor dumneavoastră pentru a crea intrarea corectă pentru acest utilizator, ca mai jos:

```
putenv("TZ=PST8PDT");
```

Următorul program, *set_tz.c*, utilizează funcția *putenv* pentru a fixa corect fusul orar. Programul utilizează apoi variabila globală *tzname* pentru a afișa valorile fusului orar, ca mai jos:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main(void)
  {
    putenv("TZ=PST8PDT");
    tzset();
    printf("Fusul orar curent este %s\n", tzname[0]);
    if (tzname[1])
      printf("Zona de orar de vara este %s\n", tzname[1]);
    else
      printf("Zona de orar de vara nu este definita\n");
  }
```

OBȚINEREA INFORMAȚIILOR DESPRE FUSUL ORAR

C/C++ 642

Câteva dintre secțiunile din acest capitol prezintă modalități prin care programele dumneavoastră obțin informații despre fusul orar. Una dintre cele mai utile funcții pe care programele dumneavoastră le pot folosi pentru a obține informații despre fusul orar este *ftime*, ca mai jos:

```
#include <sys\timeb.h>

void ftime(struct timeb *fusorar);
```

Parametrul *fusorar* este un pointer către o structură de tip *timeb*, ca mai jos:

```
struct timeb
    {
      time_t time;
      unsigned short millitm;
      short timezone;
      short dstflag;
    };
```

Câmpul *time* conține numărul de secunde de la 1 ianuarie 1970 (GMT). Câmpul *millitm* conține partea fracționară de secunde în milisecunde. Câmpul *timezone* conține diferența între ora locală și cea GMT în minute. În sfârșit, câmpul *dstflag* specifică dacă orarul de vară este în vigoare (dacă valoarea indicatorului este 1) sau nu (dacă valoarea indicatorului este 0). Următorul program, *ftime.c*, utilizează funcția *ftime* pentru a afișa informații despre fusul orar curent:

```
#include <stdio.h>
#include <time.h>
#include <sys\timeb.h>

void main(void)
  {
    struct timeb fusorar;
    tzset();
    ftime(&fusorar);
    printf("Secunde de la 1 ianuarie 1970 (GMT) %ld\n",
           fusorar.time);
    printf("Parti de secunde %d\n", fusorar.millitm);
    printf("Diferenta de ora intre GMT si cea locala %d\n",
           fusorar.timezone / 60);
    if (fusorar.dstflag)
      printf("Orarul de vara este in vigoare\n");
    else
      printf("Orarul de vara nu este in vigoare\n");
  }
```

643 FIXAREA OREI SISTEMULUI ÎN SECUNDE DE LA MIEZUL NOPȚII 1.01.1970

C/C++

Câteva dintre secțiunile din acest capitol prezintă modalități de fixare a orei sistemului, utilizând DOS și BIOS. În plus față de metodele prezentate anterior, programele dumneavoastră pot utiliza, de asemenea, funcția *stime* pentru fixarea orei sistemului în secunde de la miezul nopții 1.01.1970:

```
#include <time.h>

void stime(time_t *secunde);
```

Funcția *stime* returnează întotdeauna 0. Următorul program, *stime.c*, utilizează funcția *stime* pentru a fixa data exact cu o zi mai înainte de data și ora curente:

```
#include <time.h>

void main(void)
  {
    time_t secunde;
    time(&secunde); // Ora curenta
    secunde += (time_t) 60 * 60 * 24;
    stime(&secunde);
  }
```

644 CONVERSIA DATEI ÎN SECUNDE DE LA MIEZUL NOPȚII 1.01.1970

C/C++

Câteva secțiuni din această carte prezintă funcții de bibliotecă run-time care utilizează sau returnează secunde de la miezul nopții 1.01.1970. Pentru a vă ajuta să determinați secundele pentru o anumită dată, programele dumneavoastră pot utiliza funcția *mktime*:

```
#include <time.h>

time_t mktime(struct tm *campuri_timp);
```

Când câmpurile de timp sunt valide, funcția returnează numărul de secunde pentru respectiva oră și dată calendaristică. Dacă apare o eroare, funcția returnează –1. Parametrul *campuri_timp* este un pointer la o structură de tip *tm*, ca mai jos:

```
struct tm
    {
       int tm_sec;
       int tm_min;
       int tm_hour;
       int tm_mday;
       int tm_mon;
       int tm_year;
       int tm_wday;
       int tm_yday;
```

```
        int tm_isdst;
    };
```

Următorul program, *mktime.c*, utilizează funcția *mktime* pentru a determina numărul de secunde între miezul nopții 1.01.1970 și miezul nopții 31.10.1997:

```
#include <stdio.h>
#include <time.h>

void main(void)
   {
     time_t secunde;
     struct tm campuri_timp;
     campuri_timp.tm_mday = 31;
     campuri_timp.tm_mon = 10;
     campuri_timp.tm_year = 97;
     campuri_timp.tm_hour = 0;
     campuri_timp.tm_min = 0;
     campuri_timp.tm_sec = 0;
     secunde = mktime(&campuri_timp);
     printf("Numarul de secunde intre 1-1-70 si 31-10-97 este
       %ld\n", secunde);
   }
```

Observație: *Atunci când transmiteți o structură* ***tm*** *parțială către funcția* ***mktime****, funcția va completa câmpurile care nu sunt corecte. Funcția* ***mktime*** *acceptă date calendaristice de la 1 ianuarie 1970 până la 19 ianuarie 2028.*

DETERMINAREA DATEI ÎN CALENDAR IULIAN

C/C++645

În secțiunea 644 ați utilizat funcția *mktime* pentru a determina numărul de secunde dintre o anumită dată și miezul nopții de 1 ianuarie 1970. Așa cum ați învățat, funcția *mktime* utilizează o structură de tip *tm* pentru a păstra componentele datei calendaristice. Dacă una sau mai multe dintre componente nu sunt complete, funcția *mktime* le completează. Dacă analizați structura *tm*, veți vedea membrul *tm_yday*. Atunci când invocați funcția *mktime*, funcția va atribui membrului *tm_yday* data din calendarul iulian pentru ziua specificată. Calendarul iulian este identic cu calendarul gregorian, cu excepția faptului că el începe anul 1 la anul 46 î. Hr. din calendarul gregorian. Calculatorul redă datele din calendarul iulian în format de trei cifre. Următorul program, *iulian.c*, utilizează funcția *mktime* pentru a determina data din calendarul iulian corespunzătoare pentru 31 octombrie 1997:

```
#include <stdio.h>
#include <time.h>

void main(void)
   {
     time_t seconds;
     struct tm campuri_timp;
     campuri_timp.tm_mday = 31;
```

```
    campuri_timp.tm_mon = 10;
    campuri_timp.tm_year = 97;
    if (mktime(&campuri_timp) == -1)
      printf("Eroare la conversia campurilor\n");
    else
      printf("Data din calendarul iulian pentru
         31 octombrie 1997 este %d\n", campuri_timp.tm_yday);
 }
```

646 CREAREA UNUI ȘIR DE CARACTERE FORMATAT PENTRU DATĂ ȘI ORĂ

C/C++

Așa cum ați învățat, funcțiile *_strdate* și *_strtime* returnează data și ora curente în format șir de caractere. Multe compilatoare vă pun la dispoziție următoarea funcție *strftime* astfel ca să puteți controla mai bine formatul șirului de caractere cu dată și oră:

```
#include <time.h>

size_t strftime(char *sir, size_t lung_max, const char *format,
                const struct tm *data_ora);
```

Parametrul *sir* este un șir de caractere în care funcția *strftime* scrie șirul de caractere formatat pentru dată și oră. Parametrul *lung_max* specifică numărul maxim de caractere pe care funcția *strftime* le poate plasa în șir. Șirul *format* utilizează specificatorul de format pentru caractere *%litera* similar cu funcția *printf* pentru a specifica formatul dorit. Tabelul 646 listează caracterele valide pe care le puteți plasa în șirul de caractere formatat. În sfârșit, parametrul *data_ora* este un pointer la o structură de tip *tm* care conține câmpuri de dată și oră. Funcția *strftime* returnează o sumă a numărului de caractere atribuite parametrului *sir* sau 1 dacă funcția depășește parametrul *sir*. Tabelul 646 listează specificatorii de format pentru funcția *strftime*.

Specificator de format	Semnificație
%%	Simbolul procent %
%a	Numele abreviat al zilei din săptămână
%A	Numele complet al zilei din săptămână
%b	Numele abreviat al lunii
%B	Numele complet al lunii
%c	Data și ora
%d	Două cifre pentru ziua din lună, între 01 și 31
%H	Două cifre pentru oră, între 00 și 23
%I	Două cifre pentru oră, între 01 și 12
%j	Trei cifre pentru ziua din calendarul iulian
%m	Luna ca zecimal între 1 și 12
%M	Două cifre pentru minute, între 00 și 59

Specificator de format	Semnificație
%p	Caracterele AM sau PM
%S	Două cifre pentru secunde, între 00 și 59
%U	Două cifre pentru numărul săptămânii, de la 00 la 53, cu duminica prima zi a săptămânii
%w	Ziua din săptămână (0 = duminică, 6 = sâmbătă)
%W	Două cifre pentru numărul săptămânii, de la 00 la 53, cu luni prima zi a săptămânii
%x	Data
%X	Ora
%y	Două cifre pentru an, între 00 și 99
%Y	Patru cifre pentru an
%Z	Numele fusului orar

***Tabelul 646** Specificatorii de format pentru funcția **strftime**.*

Următorul program, *strftime.c*, ilustrează utilizarea funcției *strftime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
  {
    char buffer[128];
    struct tm *data_ora;
    time_t ora_curenta;
    tzset();
    time(&ora_curenta);
    data_ora = localtime(&ora_curenta);
    strftime(buffer, sizeof(buffer), "%x %X", data_ora);
    printf("Utilizeaza %%x %%X: %s\n", buffer);
    strftime(buffer, sizeof(buffer), "%A %B %d, %Y", data_ora);
    printf("Utilizeaza %%A %%B %%d %%Y: %s\n", buffer);
    strftime(buffer, sizeof(buffer), "%I:%M%p", data_ora);
    printf("Utilizeaza %%I:%%M%%p: %s\n", buffer);
  }
```

Atunci când compilați și executați programul *strftime.c*, ecranul dumneavoastră va afișa o ieșire similară cu următoarea (ecranul dumneavoastră va arăta o ieșire bazată pe data și ora curente):

```
Utilizeaza %x %X: 08/22/97 22:03:13
Utilizeaza %A %B %d %Y: vineri august 22 1997
Utilizeaza %I:%M%p 10:03PM
C:\>
```

647 TIPURILE DE CEASURI ALE CALCULATORULUI



Câteva dintre secțiunile acestui capitol discută despre datele calendaristice și ora PC-ului. Pentru a înțelege mai bine aceste funcții, trebuie să cunoașteți că PC-ul utilizează patru tipuri de bază de ceasuri: cronometrul, ceasul CPU, ceasul de timp real și ceasul CMOS, pe care le detaliază lista următoare:

- Cronometrul este un cip intern al PC-ului care generează o întrerupere de 18,2 ori pe secundă. La fiecare bătaie a cronometrului, PC-ul generează *interrupt 8* (un mesaj de sistem). Prin captarea acestei întreruperi, programele rezidente în memorie se pot activa singure la un anumit interval de timp.
- Ceasul CPU controlează cât de rapid se execută programele dumneavoastră. Atunci când utilizatorii spun că utilizează un sistem de 200MHz, ei se referă la ceasul CPU.
- Ceasul de timp real urmărește data și ora curente. În cele mai multe cazuri, ceasul de timp real conține aceeași valoare cu ceasul CMOS.
- Ceasul CMOS este întreținut de calculator, spre deosebire de ceasul de timp real, care este întreținut de sistemul de operare. Ceasul CMOS conține, în general, aceeași intrare cu ceasul de timp real.

648 AȘTEPTAREA APĂSĂRII UNEI TASTE



Există multe programe care, atunci când afișează un mesaj, așteaptă ca utilizatorul să apese o tastă înainte de a șterge mesajul și să continue execuția. Pentru a ajuta programele dumneavoastră să efectueze o astfel de procesare, puteți utiliza următoarea funcție *kbhit*, care returnează valoarea adevărat dacă utilizatorul apasă o tastă și fals dacă utilizatorul nu o apasă:

```
#include <conio.h>

int kbhit(void);
```

Următorul program, *kbhit.c*, va afișa un mesaj pe ecran care cere utilizatorului să apese o tastă pentru a continua. Programul utilizează apoi funcția *kbhit* pentru a aștepta intrarea de la tastatură:

```
#include <stdio.h>
#include <conio.h>

void main(void)
  {
    printf("Apasa orice tasta pentru a continua...");
    while (! kbhit());
      ;
    printf("Gata\n");
  }
```

SOLICITAREA PAROLEI DE LA UTILIZATOR

C/C++ 649

În funcţie de programele dumneavoastră, uneori este posibil să fie nevoie să cereţi utilizatorului o parolă. Atunci când utilizatorul introduce parola, intrările de la tastatură nu ar trebui să apară pe ecran. Programele dumneavoastră pot utiliza următoarea funcţie *getpass* pentru a executa această activitate:

```
#include <conio.h>
int *getpass(const char *prompt);
```

Funcţia *getpass* va afişa solicitarea specificată şi apoi va aştepta ca utilizatorul să introducă parola şi să apese apoi pe ENTER. Funcţia *getpass* returnează apoi un pointer la parola introdusă de utilizator. Următorul program, *getpass.c*, utilizează funcţia *getpass* pentru a cere utilizatorului parola:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
  {
    char *parola;
    parola = getpass("Introduceti parola:");
    if (strcmp(parola, "Bible"))
      printf("Parola incorecta\n");
    else
      printf("Parola OK\n");
  }
```

***Observaţie:** Când compilatorul nu dispune de funcţia **getpass**, puteţi utiliza funcţia **get_password**, arătată în secţiunea 650.*

SCRIEREA PROPRIEI FUNCŢII PENTRU PAROLĂ

C/C++ 650

În secţiunea 649 aţi învăţat cum se utilizează funcţia *getpass* pentru a cere utilizatorului o parolă. Aşa cum aţi învăţat, funcţia *getpass* nu afişează intrările de la tastatură ale utilizatorului. Unii utilizatori neexperimentaţi vor avea dificultăţi la introducerea parolei dacă pe ecran nu apare nici o intrare, astfel că unele programe vor afişa un asterisc (*) pentru fiecare tastă apăsată de utilizator. Pentru a cere utilizatorului o parolă şi a afişa un asterisc pentru fiecare intrare, puteţi utiliza funcţia *get_password*, cum arătăm în continuare, în cadrul programului *parola.c*:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

#define BACKSPACE 8
char *get_password(const char *prompt)
  {
    static char buffer[128];
```

```
    int i = 0;
    char litera = NULL;
    printf(prompt);
    while ((i < 127) && (litera != '\r'))
      {
        litera = getch();
        if (litera == BACKSPACE)
          {
            if (i > 0)
              {
                buffer[--i] = NULL; // sterge anteriorul *
                putchar(BACKSPACE);
                putchar(' ');
                putchar(BACKSPACE);
              }
            else
              putchar(7); // Caracter ASCII de atentionare (beep)
          }
        else if (litera != '\r')
          {
            buffer[i++] = litera;
            putchar('*');
          }
      }
    buffer[i] = NULL;
    return (buffer);
  }
void main(void)
  {
    char *parola;
    parola = get_password ("Introduceti parola: ");
    if (strcmp(parola, "Bible"))
      printf("\nParola incorecta\n");
    else
      printf("\nParola OK\n");
  }
```

651 REDIRECTAREA IEȘIRII

C/C++

De fiecare dată când executați o comandă, sistemul de operare asociază dispozitivul de intrare (*input*) implicit cu tastatura dumneavoastră. Sistemul de operare face referință la monitor ca la dispozitivul standard de ieșire (*output*) sau *stdout*. Utilizând operatorul de redirectare a ieșirii (>), puteți indica sistemului de operare să îndrepte ieșirea unui program către un fișier sau la un alt dispozitiv. Următoarea comandă, de exemplu, cere sistemului DOS să redirecteze ieșirea comenzii *dir* de la ecranul monitorului către imprimantă:

```
C:\> DIR > PRN <ENTER>
```

În mod similar, următoarea comandă indică sistemului DOS să redirecteze ieșirea comenzii *chkdsk* către fișierul *diskinfo.dat*:

```
C:\> CHKDSK > DISKINFO.DAT <ENTER>
```

Pentru a vă ajuta să scrieți programe care acceptă redirectarea ieșirii, fișierul antet *stdio.h* definește constanta *stdout* spre care operațiile de ieșire cu fișiere pot redirecta ieșirile. Câteva dintre secțiunile prezentate în această carte scriu ieșiri la *stdout*.

REDIRECTAREA INTRĂRII — C/C++ 652

De fiecare dată când executați o comandă, sistemul de operare asociază dispozitivul de ieșire implicit cu ecranul monitorului dumneavoastră. Sistemul de operare face referință la tastatură ca la dispozitivul standard de intrare sau *stdin*. Puteți utiliza operatorul de redirectare a intrării (<) pentru a indica sistemului de operare să îndrepte intrarea unui program de la tastatură către un fișier sau un alt dispozitiv. Următoarea comandă, de exemplu, indică sistemului DOS să redirecteze intrarea comenzii *more* de la tastatură către fișierul *config.sys*:

```
C:\> MORE < CONFIG.SYS <ENTER>
```

În mod similar, următoarea comandă indică sistemului DOS să redirecteze intrarea comenzii *sort* de la tastatură către fișierul *autoexec.bat*:

```
C:\> SORT < AUTOEXEC.BAT <ENTER>
```

Pentru a vă ajuta să scrieți programe care acceptă redirectarea intrării, fișierul antet *stdio.h* definește constanta *stdin* de la care operațiile de intrare cu fișiere pot obține intrările. Câteva dintre secțiunile prezentate în această carte citesc intrări de la *stdin*.

COMBINAREA REDIRECTĂRII INTRĂRII ȘI IEȘIRII — C/C++ 653

Așa cum am arătat în secțiunile 651 și 652, puteți modifica sursa implicită de intrări și ieșiri a unui program de la tastatură și monitor utilizând operatorii de redirectare a intrării (<) și ieșirii (>). Când creați o serie de programe care acceptă redirectarea intrării și ieșirii, uneori veți dori să redirectați sursele de intrare și ieșire în aceeași comandă. De exemplu, următoarea comandă indică sistemului DOS să sorteze conținutul fișierului *config.sys* și să scrie ieșirea sortată la imprimantă:

```
C:\> SORT < CONFIG.SYS > PRN <ENTER>
```

Pentru a înțelege procesul executat de sistemul de operare, citiți linia de comandă de la stânga la dreapta. Operatorul de redirectare a intrării (<) directează comanda *sort* pentru obținerea intrării sale de la fișierul *config.sys*. De asemenea, operatorul de redirectare a ieșirii (>) directează ieșirea comenzii *sort* de la monitor către imprimantă.

UTILIZAREA CONSTANTELOR STDOUT ȘI STDIN — C/C++ 654

În secțiunile 651 și 652, ați învățat că limbajul C definește indicatorii de fișier *stdin* și *stdout*. Indicatorii de fișier vă permit să scrieți programe care acceptă redirectarea I/O. Următorul

program, *majusc.c*, citește o linie de text de la indicatorul de fișier *stdin* și convertește textul în majuscule. Programul scrie apoi linia de text în *stdout*. Programul continuă cu scrierea textului până detectează sfârșitul de fișier:

```
#include <stdio.h>
#include <string.h>

void main(void)
   {
     char linie[255]; // Linie de text citita
     while (fgets(linie, sizeof(linie), stdin))
       fputs(strupr(linie), stdout);
   }
```

Utilizând comanda *upper*, puteți să afișați conținutul fișierului *autoexec.bat*, cum arătăm în continuare:

```
C:\> UPPER < AUTOEXEC.BAT <ENTER>
```

Următoarea comandă utilizează operatorul de redirectare a ieșirii pentru a tipări conținutul fișierului *config.sys* în majuscule:

```
C:\> UPPER < CONFIG.SYS > PRN <ENTER>
```

Dacă invocați comanda *upper* fără să utilizați un operator de redirectare I/O, așa cum arătăm în următorul exemplu, comanda *upper* va citi intrarea sa de la tastatură și va scrie ieșirea pe ecranul monitorului:

```
C:\> UPPER <ENTER>
```

De fiecare dată când introduceți o linie de text și apăsați pe ENTER, comanda *upper* va afișa textul corespunzător cu majuscule. Pentru a încheia programul, trebuie să apăsați combinația de taste de sfârșit de fișier, CTRL+Z (sub DOS) sau CTRL+D (sub Unix).

655 OPERATORUL PIPE



În secțiunile 651 și 652, ați învățat cum se utilizează operatorii de redirectare a intrării și ieșirii pentru a schimba sursa de intrare de la tastatură a unui program cu un fișier sau dispozitiv. Ați învățat, de asemenea, că puteți utiliza operatorii de redirectare a intrării și ieșirii pentru a îndrepta ieșirea unui program de la ecranul monitorului la un fișier sau dispozitiv. Atât sistemul DOS, cât și Unix dispun, de asemenea, de un al treilea operator de redirectare, numit operatorul *pipe* (*canal de transfer*) care vă permite să redirectați ieșirea unui program pentru a deveni intrarea altui program. De exemplu, următoarea comandă indică sistemului DOS să redirecteze ieșirea comenzii *dir* pentru a deveni intrarea pentru comanda *sort*:

```
C:\> DIR | SORT <ENTER>
```

Programele care primesc intrarea de la altă comandă sau fișier și apoi modifică intrarea într-un anumit fel, se numesc *filtre*. Următoarea comandă, de exemplu, utilizează comanda *find* pentru a filtra ieșirea comenzii *dir* și a afișa numai intrările subdirectoare:

```
C:\> DIR | FIND "<DIR>" <ENTER>
```

Tot așa cum ați utiliza mai mulți operatori de redirectare a intrării și ieșirii în aceeași linie de comandă, la fel puteți să plasați doi sau mai mulți operatori *pipe* în aceeași linie de comandă. De exemplu, următoarea comandă utilizează trei operatori *pipe* pentru a afișa numele subdirectoarelor sortate, ecran după ecran:

```
C:\> DIR | FIND "<DIR>" | SORT | MORE <ENTER>
```

FUNCȚIILE GETCHAR ȘI PUTCHAR

C/C++ 656

Multe programe utilizează funcțiile macro *getchar* și *putchar* pentru intrare și ieșire de tip caracter. De exemplu, următorul program, *minusc.c*, va converti fiecare linie de intrare a utilizatorului în minuscule și apoi va afișa fiecare linie a intrării utilizatorului pe ecran:

```
#include <stdio.h>
#include <ctype.h>   // Cuprinde prototipul lui tolower

void main(void)
  {
    int litera;
    for (litera = getchar(); ! feof(stdin); litera = getchar())
      putchar(tolower(litera));
  }
```

Următoarea comandă utilizează programul *minusc.c* pentru a tipări conținutul fișierului *autoexec.bat* cu minuscule:

```
C:\> MINUSC < AUTOEXEC.BAT > PRN <ENTER>
```

Atunci când utilizați funcțiile macro *getchar* și *putchar*, programele dumneavoastră vor accepta automat redirectarea I/O. Pentru a înțelege mai bine cum se realizează redirectarea I/O, studiați fișierul antet *stdio.h*. În cadrul fișierului *stdio.h*, veți întâlni funcțiile macro *getchar* și *putchar*, care își definesc sursa de intrare și ieșire în termenii *stdin* și *stdout*, cum arătăm în continuare:

```
#define getchar() getc(stdin)
#define putchar(c) putc((c),stdout)
```

NUMĂRAREA INTRĂRII REDIRECTATE

C/C++ 657

În funcție de conținutul unui fișier sau de ieșirea unui program, puteți să precedați fiecare linie a conținutului fișierului sau a ieșirii programului cu un număr de linie. Următorul program, *numar.c*, filtrează intrarea sa pentru a preceda fiecare linie cu numărul corespunzător de linie:

```
#include <stdio.h>

void main(void)
  {
    char linie[255];        // Linie de intrare
    long numar_linie = 0;   // Numar curent de linie
```

```
    while (fgets(linie, sizeof(linie), stdin))
      printf("%ld %s", ++numar_linie, linie);
  }
```

De exemplu, următoarea comandă tipărește o copie a fișierului *numar.c*, cu un număr de linie precedând fiecare linie:

```
C:\> NUMBER < NUMAR.C > PRN <ENTER>
```

658 SĂ NE ASIGURĂM CĂ UN MESAJ VA APĂREA PE ECRAN



Puteți utiliza operatorii de redirectare a ieșirii și operatorul pipe pentru a redirecta ieșirea programului de la ecran la un fișier, la un dispozitiv sau ca intrare la alt program. Deși o astfel de redirectare a ieșirii poate fi un instrument puternic, el poate cauza utilizatorilor pierderea unui mesaj de eroare, dacă ei nu își urmăresc cu atenție lucrul. Pentru a înțelege mai bine aceasta, să analizăm următorul program, *tipar.c*, care va afișa conținutul unui fișier pe ecran:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    char linie[255]; // Linie citita din fisier
    FILE *pointer_fisier;
    if (pointer_fisier = fopen(argv[1], "r"))
      {
        while (fgets(linie, sizeof(linie), pointer_fisier))
          fputs(linie, stdout);
        fclose(pointer_fisier);
        exit(0); // Succes
      }
    else
      {
        printf("Nu se poate deschide %s\n", argv[1]);
        exit (1);
      }
  }
```

Dacă, însă, ați dorit să trimiteți ieșirea programului către un dispozitiv sau fișier, puteți să utilizați operatorul de redirectare pentru a redirecta ieșirea programului. Următoarea comandă, de exemplu, redirectează ieșirea programului *tipar.c* pentru a tipări fișierul *autoexec.bat*:

```
C:\> TIPAR AUTOEXEC.BAT > PRN <ENTER>
```

Dacă programul reușește să deschidă fișierul *autoexec.bat*, el va scrie conținutul fișierului în *stdout*, care, pe baza operatorului de redirectare, va tipări fișierul la imprimantă. Dacă programul nu reușește să deschidă fișierul specificat, el va utiliza funcția *printf* pentru a afișa

un mesaj de eroare care afirmă că nu poate deschide fișierul. Din păcate, datorită operatorului de redirectare, mesajul nu va apărea pe ecran; el va merge către imprimantă. Este posibil ca utilizatorul să considere, în mod eronat, că execuția comenzii a reușit, dacă nu cumva verifică imediat ieșirea la imprimantă. Pentru a preveni redirectarea accidentală a mesajelor de eroare, limbajul C definește indicatorul de fișier *stderr* pe care programele dumneavoastră nu îl pot redirecta de la ecranul monitorului. Atunci când programele dumneavoastră trebuie să afișeze un mesaj de eroare, programele trebuie să utilizeze funcția *fprintf* pentru a scrie mesajul la *stderr*, cum arătăm în continuare:

```
fprintf(stderr, "Nu se poate deschide %s\n", argv[1]);
```

SCRIEREA PROPRIEI DUMNEAVOASTRE COMENZI MORE

C/C++ 659

Unul dintre cele mai bune filtre pe care sistemele DOS și Unix le pun la dispoziție este comanda *more*, care va afișa intrarea sa ecran cu ecran. De fiecare dată când comanda *more* afișează un ecran de ieșire, ea va face o pauză, așteptând ca utilizatorul să apese o tastă și va afișa apoi următorul mesaj:

```
- More -
```

Când utilizatorul apasă o tastă, comanda *more* repetă procesul și afișează următorul ecran de ieșire. Următorul program, *more.c*, implementează comanda *more*:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    char buffer[256];
    long nr_rand = 0;
    union REGS inregs, outregs;
    int ctrl_apasat, codscan;
    while (fgets (buffer, sizeof(buffer), stdin))
      {
        fputs (buffer, stdout);
        if ((++nr_rand % 24) == 0)
          {
            printf ("-- More --");
            // obtine codul de scanare al tastei apasate
            inregs.h.ah = 0;
            int86 (0x16, &inregs, &outregs);
            codscan = outregs.h.ah;
            // obtine starea tastaturii in caz de Ctrl-C
            ctrl_apasat = 0;
            inregs.h.ah = 2;
            int86 (0x16, &inregs, &outregs);
            // Indicatorul tastei Ctrl e bitul 2
```

```
            ctrl_apasat = (outregs.h.al & 4);
            // codul de scanare pentru C este 0x2E
            if ((ctrl_apasat) && (codscan == 0x2E))
            break; // Ctrl-C apasat
            printf ("\r");
          }
      }
  }
```

De fiecare dată când comanda *more* face o pauză pentru ca utilizatorul să apese o tastă, el invocă o întrerupere BIOS de tastatură (INT 16H) pentru a obține intrarea de la tastatură. Deoarece sistemul DOS definește operațiile sale de intrare în termeni de *stdin*, nu puteți utiliza *getchar*, *getch* sau *kbhit* pentru a citi intrarea de la tastatură. Funcțiile DOS de intrare utilizează următoarea intrare redirectată, de aceea vor trata următorul caracter redirectat ca apăsare de tastă a utilizatorului. Serviciile BIOS, însă, nu sunt definite în termeni de *stdin* și de aceea operatorul de redirectare nu afectează serviciile BIOS de intrare.

660 AFIȘAREA SUMEI LINIILOR REDIRECTATE



Câteva dintre secțiunile acestui capitol au creat comenzi filtru pe care le puteți utiliza cu operatorii de redirectare a intrării și operatorul pipe. Următorul program, *nr_linii.c*, va afișa suma liniilor intrării redirectate:

```
#include <stdio.h>

void main(void)
  {
    char linie[256]; // Linie de intrare redirectata
    long nr_linii = 0;
    while (fgets(linie, sizeof(linie), stdin))
      nr_linii++;
    printf("Numarul de linii redirectate: %ld\n", nr_linii);
  }
```

661 AFIȘAREA SUMEI CARACTERELOR REDIRECTATE



Câteva dintre secțiunile acestui capitol au creat comenzi filtru pe care le puteți utiliza cu operatorii DOS de redirectare a intrării și operatorul pipe. În mod similar, următorul program, *nr_carac.c*, va afișa suma caracterelor din intrarea redirectată:

```
#include <stdio.h>

void main(void)
  {
    long nr_caractere = 0;
    getchar();
    while (! feof(stdin))
      {
```

```
        getchar();
        nr_caractere++;
      }
    printf("Numarul de caractere redirectate este %ld\n",
           nr_caractere);
  }
```

CREAREA UNEI COMENZI MORE PERIODICE

C/C++ 662

Câteva dintre secțiunile acestui capitol au creat comenzi filtru pe care le puteți utiliza cu operatorii DOS de redirectare a intrării și operatorul pipe. În mod similar, următorul program, *more15.c*, schimbă comanda *more* pentru a afișa un ecran de intrări redirectate, fie la fiecare intrare de la tastatură, fie la fiecare 15 secunde, în funcție de primul eveniment care apare:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>

void main(void)
  {
    char buffer[256];
    char tasta_apasata = 0;
    long int contor = 1;
    union REGS inregs, outregs;
    time_t start_time, current_time, end_time;
    while (fgets(buffer, sizeof(buffer), stdin))
      {
        fputs (buffer, stdout);
        if ((++contor % 25) == 0)
          {
            time (&start_time);
            end_time = start_time + 15;
            do
              {
                tasta_apasata = 0;
                time (&current_time);
                inregs.h.ah = 1;
                int86 (0x16, &inregs, &outregs);
                if ((outregs.x.flags & 64) == 0)
                  {
                    tasta_apasata = 1;
                    do
                      {
                        inregs.h.ah = 0;
                        int86 (0x16, &inregs, &outregs);
                          inregs.h.ah = 1;
                          int86 (0x16, &inregs, &outregs);
```

```
                } while (! (outregs.x.flags & 64));
              }
           }
         while ((current_time != end_time) && (!tasta_apasata));
       }
    }
 }
```

663 PREVENIREA REDIRECTĂRII I/O

C/C++

Aşa cum aţi învăţat, atunci când creaţi programe care acceptă redirectarea I/O, puteţi să construiţi o bibliotecă de comenzi filtru puternice. Însă, multe programe pe care le veţi crea nu vor accepta redirectarea I/O. În funcţie de funcţiile pe care programul dumneavoastră le execută, pot apărea erori grave atunci când lăsaţi să se producă redirectarea. Următorul program, *nu_indir.c*, testează indicatoarele de fişier *stdin* şi *stdout* pentru a se asigura că ele nu vor fi redirectate:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
     union REGS inregs, outregs;
     // verifica stdin mai intai
     inregs.x.ax = 0x4400;
     inregs.x.bx = 0; // stdin are indicator 0
     intdos (&inregs, &outregs);
     if ((outregs.x.dx & 1) && (outregs.x.dx & 128))
      fprintf (stderr, "stdin nu a fost redirectat\n");
     else
      fprintf (stderr, "stdin a fost redirectat\n");
     // acum verifica stdout
     inregs.x.ax = 0x4400;
     inregs.x.bx = 1; // stdout are indicator 1
     intdos (&inregs, &outregs);
     if ((outregs.x.dx & 2) && (outregs.x.dx & 128))
      fprintf (stderr, "stdout nu a fost redirectat\n");
     else
      fprintf (stderr, "stdout a fost redirectat\n");
  }
```

Programul utilizează serviciul DOS INT 21H, funcţia 4400H, pentru a examina indicatorul de fişier. Dacă indicatorul arată un dispozitiv, atunci serviciul dă bitului 7 al registrului DX valoarea 1. Dacă serviciul stabileşte bitul 7 şi bitul 2 la valoarea 1, atunci indicatorul face referinţă la *stdout*. Dacă serviciul stabileşte bitul 7 şi bitul 1 la valoarea 1, atunci indicatorul face referinţă la *stdin*. Dacă serviciul nu dă valoarea 1 bitului 7, atunci programul a redirectat indicatorul la un fişier. Dacă serviciul nu dă valoarea 1 bitului 1 sau 2, atunci programul a redirectat indicatorul la un dispozitiv, altul decât *stdin* sau *stdout*.

Programele dumneavoastră pot utiliza serviciul INT 21H 4400H pentru a determina dacă programul curent, un program executat anterior sau utilizatorul au redirectat intrarea sau ieșirea calculatorului. În funcție de rezultatul testului, programele se pot procesa adecvat.

UTILIZAREA INDICATORULUI DE FIȘIER STDPRN

C/C++ 664

Așa cum ați învățat, fișierul antet *stdio.h* definește două indicatoare fișier – *stdin* care (în mod implicit) indică tastatura și *stdout* care indică ecranul. Dacă scrieți operații de intrare și ieșire în termeni de *stdin* și *stdout*, programele dumneavoastră vor accepta automat redirectarea I/O. În mod similar, *stdio.h* definește indicatorul de fișier *stdprn*, care indică dispozitivul imprimantă standard (PRN sau LPT1). Spre deosebire de *stdin* și *stdout*, nu puteți redirecta *stdprn*. Următorul program, *prn_echo.c*, utilizează fișierul *stdprn* pentru a imprima intrarea redirectată, pe măsură ce programul afișează ieșirea pe ecran, utilizând *stdout*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char linie[255]; // Linie de text citita
    while (fgets(linie, sizeof(linie), stdin))
      {
        fputs(linie, stdout);
        strcat(linie, "\r");
        fputs(linie, stdprn);
      }
  }
```

Următoarea linie de comandă utilizează programul *prn_echo.c* pentru a tipări și afișa lista unui director:

```
C:\> DIR | PRN_ECHO <ENTER>
```

DIVIZAREA IEȘIRII REDIRECTATE LA UN FIȘIER

C/C++ 665

Când utilizați operatorul DOS pipe pentru a redirecta ieșirea unui program și a deveni intrare a altui program, puteți să salvați o copie intermediară a ieșirii programului într-un fișier. Următorul program, *tee.c*, salvează o copie intermediară a ieșirii programului într-un fișier:

```
#include <stdio.h>

void main(void)
  {
    char buffer[256];
    while (fgets(buffer, sizeof(buffer), stdin))
      {
        fputs(buffer, stdout);
```

```
        fputs(buffer, stderr);
      }
  }
```

Comanda *tee* scrie intrarea redirectată în fișierul specificat și în *stdout*, astfel încât programul dumneavoastră poate redirecta ieșirea spre alt program. Următoarea comandă, de exemplu, utilizează comanda *tee* pentru a tipări lista nesortată a unui director înainte ca lista sortată de comanda *sort* a directorului să fie afișată pe ecran:

```
C:\> DIR | TEE PRN | SORT <ENTER>
```

666 UTILIZAREA INDICATORULUI DE FIȘIER STDAUX C/C++

Așa cum ați învățat, fișierul antet *stdio.h* definește trei indicatori de fișier –*stdin* care (în mod implicit) indică tastatura; *stdout* care indică (în mod implicit) ecranul și *stdprn* care indică întotdeauna imprimanta. Dacă scrieți operații de intrare și ieșire în termeni de *stdin* și *stdout*, programele dumneavoastră vor accepta automat redirectarea I/O. În mod similar, *stdio.h* definește indicatorul de fișier *stdaux* care indică dispozitivul auxiliar standard (AUX sau COM1). Spre deosebire de *stdin și stdout*, nu puteți să redirectați *stdaux*. Următorul program, *aux_echo.c*, utilizează indicatorul de fișier *stdaux* pentru a trimite intrarea redirectată către COM1, astfel ca programul să afișeze ieșirea pe ecran utilizând *stdout*:

```
#include <stdio.h>
#include <string.h>

void main(void)
  {
    char linie[255]; // Linie de text citita
    while (fgets(linie, sizeof(linie), stdin))
      {
        fputs(linie, stdout);
        strcat(linie, "\r");
        fputs(linie, stdaux);
      }
  }
```

Următoarea linie de comandă utilizează tipărirea la *aux_echo* (imprimanta atașată la COM1) și afișează lista unui director:

```
C:\> DIR | AUX_ECHO <ENTER>
```

667 GĂSIREA APARIȚIILOR UNUI SUBȘIR ÎN CADRUL UNEI INTRĂRI REDIRECTATE C/C++

Câteva dintre secțiunile prezentate în acest capitol au creat comenzi filtru pe care le puteți utiliza cu operatorii DOS de redirectare a intrării și pipe. Următorul program, *io_caut.c*, va afișa fiecare apariție a unui cuvânt sau expresie în cadrul unei intrări redirectate:

```
#include <stdio.h>
#include <string.h>
```

```
void main(int argc, char *argv[])
  {
    char sir[256];
    while (fgets(sir, sizeof(sir), stdin))
      if (strstr(sir, argv[1]))
        fputs(sir, stdout);
  }
```

Pentru a afişa fiecare apariţie a cuvântului *#include* în cadrul fişierului *test.c*, puteţi invoca *io_caut* ca mai jos:

```
C:\> IO_CAUT #include < TEST.C <ENTER>
```

Pentru a căuta două sau mai multe cuvinte, pur şi simplu plasaţi cuvintele între ghilimele, cum arătăm în continuare:

```
C:\> IO_CAUT "Nimeni nu este mai presus de lege"
  < CONSTITU.DAT <ENTER>
```

AFIŞAREA PRIMELOR N LINII ALE UNEI INTRĂRI REDIRECTATE

C/C++668

Câteva dintre secţiunile prezentate în acest capitol au creat comenzi filtru pe care le puteţi utiliza cu operatorii de redirectare a intrării şi operatorul pipe. Următorul program, *prim_lin.c*, afişează numărul de linii pe care îl specificaţi în cadrul liniei de comandă. În mod implicit, programul va afişa primele 10 linii ale intrării redirectate, cum arătăm în continuare:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    char linie[255]; // Linie citita din fisier
    int i,j;
    FILE *pointer_fisier;
    if (argc > 2)
      j = 10;
    else
      j = argv[2];
    if (pointer_fisier = fopen(argv[1], "r"))
      {
        for (i=0; i < j; i++)
          {
            fgets(linie, sizeof(linie), pointer_fisier);
            fputs(linie, stdout);
          }
        fclose(pointer_fisier);
      }
    else
      {
```

```
        printf("Nu se poate deschide %s\n", argv[1]);
        exit (1);
      }
  }
```

De exemplu, următoarea comandă cere ca *prim_lin* să afişeze primele 10 linii ale listei redirectate a unui director:

```
C:\> DIR | PRIM_LIN <ENTER>
```

Următoarea comandă, pe de altă parte, cere ca *prim_lin* să afişeze primele 25 de linii ale listei redirectate a unui director:

```
C:\> DIR | PRIM_LIN 25 <ENTER>
```

669 ARGUMENTELE LINIEI DE COMANDĂ



Atunci când executaţi comenzi, caracterele pe care le scrieţi după linia de comandă şi înainte de a apăsa tasta ENTER constituie linia de comandă a programului. De exemplu, următoarea linie de comandă invocă un program numit *primele* utilizând două argumente, numărul de linii de afişat şi numele fişierului dorit de utilizator:

```
C:\> PRIMELE 10 NUMEFIS.EXT <ENTER>
```

Acceptarea argumentelor în linia de comandă măreşte numărul aplicaţiilor în care vă puteţi folosi programele. De exemplu, puteţi utiliza programul *primele* pentru a afişa conţinutul unui număr nelimitat de fişiere, fără a fi necesară modificarea codului programului. Din fericire, limbajul C uşurează acceptarea argumentelor în linia de comandă. De fiecare dată când apelaţi un program în C, sistemul de operare transmite către program fiecare argument al liniei de comandă, ca pe un parametru către funcţia *main*. Pentru a accesa argumentele liniei de comandă, trebuie să declaraţi funcţia *main* ca mai jos:

```
void main(int argc, char *argv[])
  {
    // instructiunile programului
  }
```

Primul parametru, *argc*, conţine numărul de intrări distincte din linia de comandă. Să presupunem că avem următoarea linie de comandă:

```
C:\> PRIMELE 10 NUMEFIS.EXT <ENTER>
```

După această invocare a liniei de comandă, parametrul *argc* va conţine valoarea 3. Deoarece valoarea pe care compilatorul de C o atribuie lui *argc* include numele comenzii, *argc* va conţine întotdeauna o valoare mai mare sau egală cu 1. Al doilea parametru, *argv*, este o matrice de pointeri la şiruri de caractere care indică fiecare argument al liniei de comandă. În cazul liniei de comandă anterioare, elementele matricei *argv* vor fi pointeri la următoarele:

```
argv[0] contine un pointer la "PRIMELE.EXE"
argv[1] contine un pointer la "10"
argv[2] contine un pointer la "NUMEFIS.EXT"
argv[3] contine NULL
```

Multe programe din această carte utilizează frecvent argumente în linia de comandă.

AFIŞAREA NUMĂRULUI DE ARGUMENTE ALE LINIEI DE COMANDĂ

C/C++ 670

De fiecare dată când invocaţi un program în C, sistemul de operare transmite numărul argumentelor liniei de comandă – ca şi pointerii către respectivele elemente – către funcţia *main*. Următorul program, *cmd_cnt.c*, utilizează parametrul *argc* pentru a afişa suma argumentelor liniei de comandă transmise programului:

```
#include <stdio.h>

void main(int argc, char *argv[])
  {
   printf ("Numarul de intrari ale liniei de comanda este %d\n",
            argc);
  }
```

Presupunând că invocaţi *cmd_cnt* fără parametri, *cmd_cnt* va afişa următoarele:

```
C:\> CMD_CNT <ENTER>
Numarul de intrari al liniei de comanda este 1
C:\>
```

Dacă includeţi argumentele *A*, *B* şi *C* ale liniei de comandă, *cmd_cnt* va afişa următoarele:

```
C:\> CMD_CNT A B C <ENTER>
Numarul de intrari al liniei de comanda este 4
C:\>
```

AFIŞAREA LINIEI DE COMANDĂ

C/C++ 671

Aşa cum aţi învăţat, de fiecare dată când invocaţi un program în C, sistemul de operare transmite numărul argumentelor liniei de comandă – ca şi pointerii la respectivele elemente – către funcţia *main*. Următorul program, *o_cmd.c* utilizează parametrul *contor* în cadrul buclei *for* pentru a afişa fiecare intrare a liniei de comandă:

```
#include <stdio.h>

void main(int argc, char *argv[])
  {
   int i;
   for (i = 0; i < argc; ++i)
     printf ("argv[%d] indica %s\n", i, argv[i]);
  }
```

Dacă invocaţi *o_cmd* fără parametri, *o_cmd* va afişa următoarele:

```
C:\> O_CMD <ENTER>
argv[0] indica C:\O_CMD.EXE
C:\>
```

De asemenea, dacă invocaţi *o_cmd* cu argumentele *A*, *B* şi *C* în linia de comandă, *o_cmd* va afişa următoarele:

```
C:\> O_CMD A B C <ENTER>
argv[0] indica C:\O_CMD.EXE
argv[1] indica A
argv[2] indica B
argv[3] indica C
C:\>
```

672 LUCRUL CU ARGUMENTE ALE LINIEI DE COMANDĂ PLASATE ÎNTRE GHILIMELE



De fiecare dată când invocaţi un program în C, sistemul de operare transmite funcţiei *main* ca parametri numărul de intrări ale liniei de comandă şi o matrice de pointeri către intrări. Pot exista ocazii când programele dumneavoastră trebuie să lucreze cu parametri pe care sistemul de operare îi transmite de la linia de comandă încadraţi între ghilimele. De exemplu, să presupunem că un program numit *cauttext* caută în fişierul precizat de utilizator un anumit text, cum arătăm în continuare:

```
C:\> CAUTTEXT "Nimeni nu este mai presus de lege"
  NUMEFIS.EXT <ENTER>
```

Majoritatea compilatoarelor de C tratează parametrii dintre ghilimele ca pe un singur argument. Încercaţi cu programul *o_cmd*, pe care l-aţi scris în secţiunea 671, pentru a determina cum tratează compilatorul parametrii dintre ghilimele:

```
C:\> O_CMD "Nimeni nu este mai presus de lege" NUMEFIS.EXT
<ENTER>
argv[0] indica O_CMD.EXE
argv[1] indica Nimeni nu este mai presus de lege
argv[2] indica NUMEFIS.EXT
C:\>
```

673 AFIŞAREA CONŢINUTULUI UNUI FIŞIER DIN LINIA DE COMANDĂ



Câteva dintre secţiunile anterioare v-au arătat cum se utilizează parametrii *argc* şi *argv* pentru a accesa parametrii liniei de comandă. Următorul program, *unfisier.c*, utilizează *argv* pentru a afişa conţinutul uni fişier specificat în linia de comandă:

```
#include <stdio.h>

void main(int argc, char *argv[])
  {
    FILE *pointer_fisier; // Pointer fisier
    char linie[255]; // Linie din fisier
    if ((pointer_fisier = fopen(argv[1], "r")) == NULL)
      printf("Eroare la deschiderea %s\n", argv[1]);
    else
      {
```

```
      // Citeste si afiseaza fiecare linie a fisierului
      while (fgets(linie, sizeof(linie), pointer_fisier))
        fputs(linie, stdout);
      fclose (pointer_fisier); // Inchide fisierul
    }
  }
```

Pentru a afişa conţinutul unui fişier, invocaţi *unfisier* cu numele de fişier dorit, cum arătăm în continuare:

```
C:\> UNFISIER NUMEFIS.EXT <ENTER>
```

Observaţi instrucţiunea *if* care deschide fişierul precizat în linia de comandă. Apelarea funcţiei *fopen* în cadrul instrucţiunii *if* încearcă să deschidă fişierul pe care l-a specificat *argv[1]*. Dacă fişierul nu există, *funcţia fopen* returnează *NULL* şi programul va afişa un mesaj în care afirmă că nu poate deschide fişierul. Dacă utilizatorul nu specifică numele fişierului, atunci *argv[1]* va conţine *NULL*, ceea ce va determina ca şi funcţia *fopen* să returneze *NULL*. Dacă funcţia *fopen* reuşeşte să deschidă fişierul, atunci programul va utiliza bucla *while* pentru a citi şi afişa conţinutul fişierului.

TRATAREA LUI ARGV CA UN POINTER

C/C++ 674

Câteva dintre secţiunile anterioare au utilizat matricea de pointeri *argv* pentru a accesa argumentele liniei de comandă. Deoarece *argv* este o matrice, programele dumneavoastră pot utiliza un pointer pentru a accesa elementele sale. Dacă utilizaţi un pointer pentru a accesa elementele lui *argv*, *argv* va deveni un pointer la o matrice de pointeri. Următorul program, *argv_ptr.c*, va trata *argv* ca un pointer la un pointer, apoi va utiliza *argv* pentru a afişa linia de comandă:

```
#include <stdio.h>

void main(int argc, char **argv)
  {
    while (*argv)
      printf ("%s\n", *argv++);
  }
```

Observaţi în program declararea lui *argv* ca pointer la un pointer la un şir de caractere. Programul utilizează instrucţiunea *while* pentru a cicla prin argumentele liniei de comandă până când valoarea indicată de **argv* este *NULL*. Dacă îl veţi apela din nou, compilatorul de C utilizează *NULL* pentru a indica ultimul argument al liniei de comandă. În cadrul buclei *while*, instrucţiunea *printf* va afişa un şir de caractere indicat de *argv*. Instrucţiunea *printf* va mări apoi valoarea lui *argv*, astfel că *argv* indică următorul argument al liniei de comandă.

CUM CUNOAŞTE COMPILATORUL DE C LINIA DE COMANDĂ

C/C++ 675

De fiecare dată când executaţi un program, sistemul de operare încarcă programul în memorie. În cazul sistemului de operare DOS, acesta încarcă mai întâi 256 de octeţi în

memorie, reprezentând *segmentul prefix al programului*, care conține informații de genul: tabelul de fișiere ale programului, segmentul de mediu și linia de comandă. Figura 675 ilustrează formatul segmentului prefix al programelor DOS.

Offset	Conținut
0H	Instrucțiunea Int 20H
2H	Vârful adresei de segment a memoriei
4H	Rezervat
5H	Apelare *far* a dispecerului DOS
AH	Vectorul Int 22H
EH	Vectorul Int 23H
12H	Vectorul Int 24H
16H	Rezervat
2CH	Adresa segmentului pentru copia mediului
2EH	Rezervat
5CH	FCB 1 implicit
6CH	FCB 2 implicit
7CH	Rezervat
80H	Lungimea în octeți a liniei de comandă
81H	Linia de comandă
FFH	

Figura 675 *Segmentul prefix al programelor DOS.*

După cum vedeți, începând de la deplasamentul 80H, DOS stochează până la 128 de octeți de informații ale liniei de comandă. Atunci când compilați un program în C, compilatorul de C adaugă un cod suplimentar care analizează informațiile liniei de comandă, atribuind codul către matricea *argv*, ceea ce face ușor de accesat aceste argumente din programele dumneavoastră în C.

676 MEDIUL



După cum știți, atât DOS, cât și Unix stochează informațiile în regiunea de memorie numită mediu. Utilizând comanda SET, puteți să afișați, să adăugați sau să modificați intrările de mediu. Conform funcției programului dumneavoastră, uneori va trebui să accesați informațiile conținute de mediu. De exemplu, multe programe utilizează intrarea de mediu TEMP pentru a determina unitatea de disc și subdirectoarea în cadrul cărora programul trebuie să creeze fișierele temporare. C face foarte ușor accesul la conținutul intrărilor de mediu. O cale de acces la mediu este declararea funcției *main*, cum arătăm în continuare:

```
void main(int argc, char *argv[], char *env[])
```

La fel cum C vă permite să utilizați o matrice de pointeri la șiruri de caractere pentru a accesa argumentele liniei de comandă a programului, puteți accesa intrările de mediu în același mod. Următorul program, *mediu.c*, utilizează matricea *env* pentru a afișa intrările curente de mediu:

```
#include <stdio.h>

void main(int argc, char *argv[], char *env[])
  {
   int i;
```

```
    for (i = 0; env[i] != NULL; i++)
      printf ("env[%d] indica %s\n", i, env[i]);
  }
```

După cum vedeți, programul ciclează prin intrările matricei *env* până găsește valoarea *NULL*, care indică programului că a găsit sfârșitul mediului.

TRATAREA MATRICEI ENV CA POINTER

C/C++ 677

În secțiunea 676 ați învățat că C vă permite utilizarea matricei de pointeri la șiruri de caractere *env* pentru a accesa conținutul mediului. Deoarece *env* este o matrice, puteți să îl tratați ca un pointer. Următorul program, *env_ptr.c*, va trata *env* ca pointer la un pointer la un șir de caractere, apoi va utiliza *env* pentru a afișa conținutul mediului:

```
#include <stdio.h>

void main(int argc, char **argv, char **env)
  {
    while (*env)
      printf("%s\n", *env++);
  }
```

După cum se vede, programul buclează până când *env* indică valoarea NULL, ceea ce înseamnă sfârșitul mediului. În cadrul buclei, instrucțiunea *printf* tipărește șirul indicat de *env* și apoi incrementează *env* pentru a indica următoarea intrare.

UTILIZAREA CUVÂNTULUI CHEIE VOID CA PARAMETRU LA FUNCȚIA MAIN

C/C++ 678

Atunci când programele dumneavoastră nu utilizează parametri în linia de comandă și nu aveți nevoie să utilizați *argc* și *argv*, puteți să omiteți parametri și să declarați funcția *main* ca mai jos:

```
void main()
```

Totuși, când o funcție nu primește parametri, trebuie să utilizați cuvântul cheie *void* pentru a face absolut clar pentru cititor că funcția nu primește parametri, cum arătăm în continuare:

```
void main(void)
```

LUCRUL CU NUMERE ÎN LINIA DE COMANDĂ

C/C++ 679

Pe măsură ce creați programe care utilizează argumente ale liniei de comandă, va trebui în cele din urmă să lucrați cu numere în linia de comandă. De exemplu, următoarea linie de comandă indică programului *primele* să afișeze primele 15 linii ale fișierului *autoexec.bat*:

```
C:\> PRIMELE 15 AUTOEXEC.BAT <ENTER>
```

Atunci când linia de comandă conține numere, matricea *argv* stochează numerele în format ASCII. Pentru a utiliza numărul, trebuie să convertiți mai întâi numărul din formatul ASCII

într-o valoare întreagă sau în virgulă mobilă. Pentru a converti numărul, utilizaţi funcţiile *atoi*, *atol* şi *atof* despre care aţi învăţat în capitolul despre funcţii matematice. Următorul program, *beep.c*, face ca difuzorul încorporat al calculatorului să emită un număr de sunete specificat în linia de comandă. De exemplu, următoarea linie de comandă cere ca *beep* să emită sunetul de trei ori:

```
C:\> BEEP 3 <ENTER>
```

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    int nr; // Numarul de sunete pe care le va emite difuzorul
    int i; // Numarul de sunete emise de difuzor
    // Determinarea numarului de sunete
    nr = atoi(argv[1]);
    for (i = 0; i < nr; i++)
     putchar(7);     // valoarea ASCII 7 face ca difuzorul
                     // sa emita un sunet
  }
```

Dacă utilizatorul specifică în linia de comandă un parametru care nu este un întreg valid, atunci funcţia *atoi* va returna valoarea 0.

680 VALORILE DE STARE DE IEŞIRE



Multe comenzi DOS acceptă valorile de stare de ieşire cu care puteţi testa reuşita comenzilor din cadrul fişierelor de comenzi (*batch*). De exemplu, comanda DOS XCOPY acceptă valorile de stare de ieşire listate în tabelul 680.

Valori de ieşire	Semnificaţie
0	Operaţie de copiere reuşită
1	Nu a găsit nici un fişier pentru a fi copiat
2	Copierea fişierului terminată de utilizator prin CTRL+C
4	Eroare de iniţializare
5	Eroare la scriere pe disc

Tabelul 680 *Valorile de stare de ieşire pentru XCOPY.*

Prin utilizarea comenzii IF ERRORLEVEL, fişierele dumneavoastră de comenzi pot testa starea la ieşire a programului pentru a verifica reuşita comenzii şi apoi continuă procesarea fişierului de comenzi în mod corespunzător. Când creaţi programe, puteţi să obţineţi valoarea stării de ieşire. Cea mai uşoară cale de a returna o valoare de stare de ieşire este utilizarea funcţiei *exit*:

```
exit(valoare_stare_iesire);
```

Următoarea apelare a funcţiei, de exemplu, returnează valoarea de stare de ieşire 1:

```
exit(1);
```

Atunci când programul dumneavoastră invocă funcția *exit*, programul se încheie imediat și returnează sistemului de operare valoarea de stare de ieșire specificată. Următorul program, *fisier.c*, va afișa conținutul unui fișier. Dacă programul nu poate deschide fișierul pe care îl specifică linia de comandă, programul va returna valoarea de stare de ieșire 1. Dacă *fisier* reușește să afișeze conținutul fișierului, atunci va returna valoarea de stare de ieșire 0:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    char linie[255]; // Linie citita din fisier
    FILE *pointer_fisier;
    if (pointer_fisier = fopen(argv[1], "r"))
      {
        while (fgets(linie, sizeof(linie), pointer_fisier))
          fputs(linie, stdout);
        fclose(pointer_fisier);
        exit(0); // Succes
      }
    else
      {
        printf("Nu poate deschide %s\n", argv[1]);
        exit (1);
      }
  }
```

Observație: Multe compilatoare de C dispun de o funcție numită ***_exit*** *care, la fel ca* ***exit****, încheie imediat un program și returnează valoarea de stare la ieșire. Însă, spre deosebire de* ***exit****, care mai întâi închide fișierele deschise și golește bufferele de ieșire, funcția* ***_exit*** *nu închide fișierele deschise, rezultând posibile pierderi de date.*

UTILIZAREA INSTRUCȚIUNII RETURN PENTRU PROCESAREA STĂRII DE IEȘIRE

C/C++681

În cadrul funcțiilor C, instrucțiunea *return* încheie execuția unei funcții și returnează valoarea specificată către funcția care a apelat-o. În cadrul funcției C *main*, instrucțiunea *return* are un comportament similar cu cel din cadrul unei funcții, încheind execuția programului și returnând valoarea specificată de program către sistemul de operare (care a apelat programul). Următorul program, *ret_exit.c*, va returna starea de ieșire 1. Dacă *ret_exit* afișează cu succes conținutul fișierului, va returna valoarea de stare de ieșire 0:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
  {
    char linie[255]; // Linie citita din fisier
    FILE *pointer fisier;
```

```
    if (pointer_fisier = fopen(argv[1], "r"))
      {
        while (fgets(linie, sizeof(linie), pointer_fisier))
          fputs(linie, stdout);
        fclose(pointer_fisier);
        return(0); // Succes
      }
    else
      {
        printf("Nu poate deschide %s\n", argv[1]);
        return(1);
      }
  }
```

Observați că programul a modificat definiția lui *main* pentru a arăta că funcția va returna o valoare întreagă pentu starea de ieșire.

682 CÂND DECLARĂM FUNCȚIA MAIN CA VOID C/C++

Câteva dintre programele prezentate de această carte definesc funcția *main* cum am arătat în fiecare dintre următoarele două implementări:

```
void main(void)
void main(int argc, char *argv[])
```

Cuvântul cheie *void* care apare înaintea lui *main* arată compilatorului de C (și programatorilor care vă citesc codul) că funcția *main* nu utilizează instrucțiunea *return* pentru a returna o valoare de stare de ieșire către sistemul de operare. Cuvântul cheie *void* nu împiedică, totuși, programele dumneavoastră să utilizeze funcția *exit* pentru a returna o valoare de stare de ieșire. De regulă, însă, dacă funcția *main* nu utilizează instrucțiunea *return*, trebuie să utilizați cuvântul cheie *void* pentru a preceda numele funcției. Dacă nu utilizați cuvântul cheie *void*, unele compilatoare pot afișa un mesaj de avertizare similar cu următorul:

```
Warning Function should return a value in function main
```

683 CĂUTAREA UNEI ANUMITE INTRĂRI ÎN MEDIU C/C++

În secțiunea 676 ați învățat cum se utilizează matricea de pointeri la șiruri de caractere *env* pentru a accesa o copie a mediului programului:

```
void main(int argc, char *argv[], char *env[])
```

Atunci când programele dumneavoastră trebuie să caute o anumită intrare de mediu, puteți considera convenabil să utilizați funcția *getenv*, care se implementează în modul următor:

```
#include <stdlib.h>
char *getenv(const char *nume_intrare);
```

Funcţia *getenv* caută o anumită intrare din intrările mediului, cum ar fi TEMP. Numele intrării nu trebuie să includă un semn egal. Dacă intrarea specificată este în mediu, funcţia *getenv* va returna un pointer la valoarea intrării. Dacă programul nu găseşte intrarea, funcţia *getenv* va returna *NULL*. Următorul program, *calea.c*, caută în mediu intrarea PATH. Dacă programul găseşte intrarea, el va afişa valoarea intrării:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    char *intrare;
    intrare = getenv("PATH");
    if (*intrare)
      printf("PATH=%s\n", intrare);
    else
      printf("PATH nu este definit\n");
  }
```

CUM ESTE TRATAT MEDIUL ÎN SISTEMUL DOS C/C++ 684

Atunci când lucraţi sub DOS, mediul vă pune la dispoziţie o regiune de memorie în care puteţi să plasaţi informaţii de configuraţie, cum ar fi calea comenzilor sau promptul sistemului. Comanda SET vă permite afişarea, adăugarea sau modificarea intrărilor de mediu:

```
C:\> SET <ENTER>
COMSPEC=C:\DOS\COMMAND.COM
PROMPT=$P$G
PATH=C:\DOS\;C:\WINDOWS;C:\TCLITE\BIN
C:\>
```

Sistemul DOS menţine o copie principală a mediului pe care utilizatorul poate să o modifice numai cu comanda SET. Atunci când invocaţi un program, sistemul DOS face o copie a conţinutului curent al mediului şi transmite copia către programul dumneavoastră, cum arătăm în figura 684.

Deoarece programele dumneavoastră primesc o copie a mediului, modificările făcute de program intrărilor de mediu nu afectează copia principală. Câteva dintre secţiunile din acest capitol prezintă funcţii care redau sau stabilesc mediul. În fiecare caz, aceste funcţii accesează numai copia de mediu a programului.

***Observaţie:** Majoritatea programelor din Windows folosesc registrul de sistem (Registry) din Windows pentru a obţine informaţii despre mediu. Veţi afla mai multe despre Registry în secţiunile următoare.*

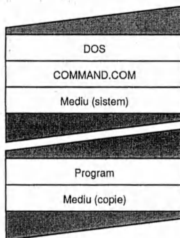


Figura 684 *DOS transmite fiecărui program propria sa copie de mediu.*

685 UTILIZAREA VARIABILEI GLOBALE ENVIRON

C/C++

În secțiunea 676 ați învățat că programele dumneavoastră pot utiliza matricea de pointeri la șiruri de caractere pe care sistemul DOS o transmite funcției *main* a programului pentru a accesa copia de mediu DOS a programului:

```
void main(int argc, char *argv[], char *env[])
```

În plus față de utilizarea matricei *env*, C definește, de asemenea, o variabilă globală numită *environ*, care conține copia de mediu a programului. Următorul program, *environ.c*, utilizează variabila globală *environ* pentru a afișa intrările de mediu:

```
#include <stdio.h>
#include <dos.h>
void main(void)
  {
    int i;
    for (i = 0; environ[i]; i++)
      printf("%s\n", environ[i]);
  }
```

Atunci când programele dumneavoastră utilizează funcția *putenv* pentru a adăuga sau modifica o intrare de mediu, ar trebui să accesați mai târziu intrările mediului utilizând funcția *getenv* sau prin accesarea variabilei globale *environ*. Pentru a plasa o intrare în copia de mediu a programului, funcția *putenv* poate să necesite mutarea copiei de mediu a programului, care invalidează pointerul transmis de DOS către funcția *main*.

ADĂUGAREA UNEI INTRĂRI LA MEDIUL CURENT

C/C++ 686

În secţiunea 684 aţi învăţat că DOS reţine o copie principală a mediului. Sistemul DOS copiază mediul principal şi transmite copia către fiecare program invocat. Ca urmare, programele dumneavoastră nu pot modifica de obicei intrările din mediul principal. În schimb, sistemul DOS permite orice modificare făcută mediului de program, numai în copia de mediu a programului. Există ocazii, însă, când programele dumneavoastră trebuie totuşi să păstreze o intrare în copia mediului. De exemplu, să presupunem că un program generează un proces copil care trebuie să cunoască numele unui anumit fişier. Programul poate să plaseze mai întâi numele fişierului într-o copie de mediu. Atunci când programul generează procesul copil, procesul copil va primi o copie a mediului programului şi deci va avea acces la numele fişierului. Pentru asemenea cazuri, programele dumneavoastră pot utiliza funcţia *putenv*:

```
#include <stdlib.h>
char putenv(const char *intrare);
```

Dacă funcţia *putenv* reuşeşte să adauge intrarea la copia de mediu a programului, ea va returna valoarea 0. Dacă apare o eroare (de exemplu, mediul este complet), atunci funcţia *putenv* va returna –1. Următorul program, *putenv.c*, ilustrează modul de utilizare a funcţiei *putenv*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    if (putenv("CARTE=Jamsa\'s C & C++ Programmer\'s Bible"))
      printf("Eroare la scrierea in mediu\n");
    else
      {
        int i;
        for (i = 0; environ[i]; ++i)
          printf("%s\n", environ[i]);
      }
  }
```

Observaţie: *Nu invocaţi funcţia **putenv** cu o variabilă şir de caractere automatică sau un pointer la şir de caractere pe care codul programului dumneavoastră poate să îl elibereze. De asemenea, dacă programele dumneavoastră utilizează matricea de pointeri la şiruri de caractere pe care sistemul DOS îl transmite funcţiei **main**, fiţi conştient că funcţia **putenv** poate să mute copia de mediu a programului, invalidând astfel pointerul **env**. Pentru a accesa intrările de mediu, utilizaţi funcţia **getenv** sau variabila globală **environ**.*

687 ADĂUGAREA ELEMENTELOR LA MEDIUL DOS



Ați învățat că atunci când rulați un program, DOS copiază intrările de mediu curent și transmite copia către program. Deoarece programul nu are acces la copia principală a intrărilor de mediu, DOS nu poate modifica intrările menținute de el. Deoarece DOS nu dispune de o protecție a memoriei care să prevină accesarea memorie unui program de către un alt program, puteți să scrieți un program care localizează și găsește copia principală de mediu DOS. După ce cunoașteți localizarea mediului, puteți să modificați sau să ștergeți o intrare existentă sau să adăugați o intrare nouă la mediu. Deoarece mediul are o dimensiune fixă, uneori este posibil ca intrările pe care vreți să le adăugați să nu aibă loc în mediu. În unele cazuri, puteți să modificați dimensiunea mediului și să alocați mai multă memorie pentru a evita depășirea spațiului.

Procesul de localizare și dimensionare a copiei principale a mediului DOS este foarte complex. Deoarece explicația și exemplele de programe ar cere 20-30 de pagini și ar depăși domeniul acestei cărți, trebuie să aflați numai că este posibil ca programele dumneavoastră să actualizeze copia principală de mediu.

688 OPRIREA PROGRAMULUI CURENT



Pe măsură ce programele dumneavoastră devin tot mai complexe, uneori, datorită apariției unor erori critice, veți dori ca programul să se încheie imediat și să afișeze un mesaj de eroare în *stderr*. În astfel de cazuri, programele dumneavoastră pot utiliza funcția *abort*:

```
#include <stdlib.h>
void abort(void);
```

Atunci când programele dumneavoastră invocă funcția *abort*, aceasta va afișa următorul mesaj în *stderr* și apoi va invoca funcția DOS *_exit* cu valoarea de stare de ieșire 3:

```
Abnormal program termination
```

Cea mai bună cale de a înțelege funcția *abort* este studierea următoarei implementări:

```
void abort(void)
  {
    fputs("Abnormal program termination", stderr);
    _exit(3);
  }
```

Este important de observat că funcția *abort* invocă funcția *_exit* și nu *exit*. Cum am discutat în secțiunea 680, funcția *_exit* nu închide orice fișier și nu golește ieșirea.

689 DEFINIREA FUNCȚIILOR CARE SE EXECUTĂ LA ÎNCHEIEREA PROGRAMULUI



Programele dumneavoastră pot să execute automat una sau mai multe funcții când programul se încheie. Pentru a face acest lucru, programul dumneavoastră poate utiliza

funcția *atexit*, care permite programului dumneavoastră să specifice până la 32 de funcții pe care programul le execută automat când se încheie:

```
#include <stdlib.h>

int atexit(void (*functie(void)));
```

Funcțiile pe care *atexit* le invocă nu pot utiliza parametri. Dacă funcțiile trebuie să acceseze anumite date, atunci trebuie să declarați datele ca variabile globale. Atunci când definiți o listă de funcții pentru încheierea programului, programul va executa funcțiile în ordine, începând cu ultima funcție înregistrată și sfârșind cu prima funcție pe care ați înregistrat-o. Următorul program, *atexit.c*, utilizează funcția *atexit* pentru a înregistra două funcții de încheiere a programului:

```
#include <stdio.h>
#include <stdlib.h>

void prima(void)
  {
    printf("Prima functie inregistrata\n");
  }
void a_doua(void)
  {
    printf("A doua functie inregistrata\n");
  }
void main(void)
{
    atexit(prima);
    atexit(a_doua);
}
```

Atunci când compilați și executați programul *atexit.c*, ecranul dumneavoastră va afișa următorul rezultat:

```
A doua functie inregistrata
Prima functie inregistrata
C:\>
```

După cum vedeți, funcțiile din lista de încheiere a programului se execută în ordinea inversă ordinii în care programul a înregistrat funcțiile (cu alte cuvinte, ultima funcție înregistrată este prima executată).

Observație: *Dacă programul dumneavoastră invocă funcția* ***exit****, programul execută funcțiile din lista de încheiere. Totuși, dacă programul dumneavoastră invocă funcția* ***_exit****, atunci programul nu execută funcțiile din lista de încheiere.*

BIBLIOTECILE

C/C++ 690

Multe dintre secțiunile din această carte se referă la funcțiile bibliotecii run-time de C. Atunci când utilizați o funcție de bibliotecă run-time în programul dumneavoastră, editorul de legături (*linker*) încarcă codul corespunzător din fișierul bibliotecă în programul dumneavoastră executabil. Avantajul evident al utilizării funcțiilor de bibliotecă run-time este acela

că nu trebuie să scrieţi respectivul cod. Dacă examinaţi fişierele care însoţesc compilatorul dumneavoastră, veţi găsi multe care au extensia LIB. Aceste fişiere conţin biblioteci obiect. Atunci când compilaţi şi editaţi legăturile programelor dumneavoastră, editorul de legături examinează fişierele LIB pentru a rezolva referinţele la funcţii.

Când creaţi funcţii utile, puteţi să construiţi propriile dumneavoastră biblioteci. Puteţi astfel să utilizaţi rapid o funcţie pe care aţi creat-o pentru un program, în cadrul altui program. Majoritatea compilatoarelor dispun de un program de bibliotecă care vă permite să construiţi şi să modificaţi biblioteci. În timp ce programul *Turbo C++ Lite* nu este capabil să creeze biblioteci, programul *Turbo C++* Borland de bibliotecă este numit TLIB, cel *C/C++* Microsoft e numit LIB şi compilatoarele de Windows C++ ale ambelor companii acceptă programe de bibliotecă. Câteva dintre următoarele secţiuni vor prezenta operaţii de bibliotecă.

691 REUTILIZAREA UNUI COD OBIECT

C/C++

Pe măsură ce creaţi funcţii, veţi găsi adesea că o funcţie scrisă de dumneavoastră pentru un program va rezolva cerinţele unui al doilea program. De exemplu, programul *dim_sir.c* (care se găseşte în CD-ROM-ul care însoţeşte această carte sub numele *str_len.c*) conţine o funcţie *dim_sir*:

```
int dim_sir(char *sir)
  {
    int dim = 0;
    while (*sir++)
      dim++;
    return(dim);
  }
```

Compilaţi fişierul pentru a crea fişierul obiect *dim_sir.obj*. După ce compilaţi fişierul obiect, scrieţi programul *dimens.c*, aşa cum arătăm în continuare, care utilizează funcţia pentru a afişa dimensiunea câtorva şiruri de caractere:

```
#include <stdio.h>

int dim_sir(char *);
void main(void)
  {
    char *titlu = "Totul despre C/C++";
    char *sectiune = "Instrumente";
    printf("Dimensiunea lui %s e %d\n", titlu, dim_sir(titlu));
    printf("Dimensiunea lui %s este %d\n", sectiune,
           dim_sir(sectiune));
  }
```

Dacă utilizaţi *Turbo C++ Lite*, compilaţi programul cum urmează, pentru ca el să utilizeze conţinutul fişierului obiect *dim_sir.obj* (puteţi, de asemenea, să creaţi un proiect şi să adăugaţi ambele fişiere *C* la proiect, cum se arată în secţiunea 268):

```
C:\> TC DIMENS.C DIM_SIR.OBJ <ENTER>
```

În acest exemplu, puteți să combinați codul obiect al funcției cu programul dumneavoastră pentru a rezolva și utiliza funcția. Așa cum veți învăța în secțiunea 692, însă, compilând fișierele obiect în acest mod nu prezintă o utilitate deosebită.

PROBLEME CU COMPILAREA FIȘIERELOR C ȘI OBJ

C/C++692

În secțiunea 691 ați învățat că pentru a reutiliza funcții, puteți compila funcția separat pentru a crea un fișier OBJ și mai târziu compilați codul programului în C și fișierul OBJ al funcției pentru a produce un program executabil:

```
C:\> TC UNFISIER.C FUNCTIE.OBJ <ENTER>
```

Deși această tehnică vă permite să rezolvați și să utilizați codul funcției, ea restricționează numărul de funcții pe care puteți să le rezolvați, la lungimea liniei de comandă. De exemplu, să presupunem că programul dumneavoastră folosește 10 funcții care se află în fișiere obiect separate. Veți constata că devine dificil să vă amintiți care fișiere trebuie compilate – chiar dacă toate numele fișierelor sunt trecute în linia de comandă. O soluție este gruparea tuturor funcțiilor dumneavoastră într-un singur fișier obiect. O soluție mai bună, însă, este să creați un fișier în care să se afle codul obiect pentru fiecare funcție. Fișierul bibliotecă este de preferat fișierului OBJ pentru că, așa cum ați învățat, majoritatea programelor de bibliotecă vă permit actualizarea rapidă a bibliotecii (prin înlocuire, adăugare sau ștergere de fișiere obiect).

CREAREA UNUI FIȘIER BIBLIOTECĂ

C/C++693

În funcție de compilatorul dumneavoastră, numele programului de bibliotecă și opțiunile liniei de comandă pe care programul dumneavoastră le acceptă vor diferi. Totuși, următoarea listă detaliază operațiile pe care majoritatea programelor de bibliotecă le acceptă:

1. Crearea unei biblioteci
2. Adăugarea unuia sau a mai multor fișiere obiect la bibliotecă
3. Înlocuirea unui fișier în cod obiect cu un altul
4. Ștergerea unuia sau mai multor fișiere obiect din bibliotecă
5. Listarea rutinelor pe care le conține biblioteca

De exemplu, următoarea comandă utilizează programul TLIB de bibliotecă al firmei Borland pentru a crea o bibliotecă numită *bibl.lib* și a insera codul obiect în bibliotecă pentru funcția *dim_sir* conținută în *dim_sir.obj*:

```
C:\> TLIB BIBL.LIB + DIM_SIR.OBJ <ENTER>
```

După ce fișierul bibliotecă este creat, puteți să utilizați biblioteca pentru a compila și lega programul *dimens.c*:

```
C:\> TC DIMENS.C BIBL.LIB <ENTER>
```

694 OPERAȚIILE OBIȘNUITE DE BIBLIOTECĂ



În funcție de compilatorul dumneavoastră, operațiile pe care le acceptă programul dumneavoastră de bibliotecă pot diferi. Însă, cele mai multe programe de bibliotecă vă permit adăugarea și eliminarea fișierelor în cod obiect, utilizând simbolul plus (+) pentru a adăuga un fișier și simbolul minus (–) pentru a elimina un fișier obiect. Tabelul 694 listează câteva operații de bibliotecă ce utilizează fișierul bibliotecă *info_mea.lib*.

Comanda	Operația
info_mea.lib +strcpy.obj	Adaugă fișierul obiect *strcpy.obj* la bibliotecă
info_mea.lib ++strlen.obj+strupr.obj	Adaugă fișierul obiect *strlen.obj* și *strupr.obj* la bibliotecă
info_mea.lib –strlwr.obj	Elimină fișierul obiect *strlwr.obj* din bibliotecă
info_mea.lib –strlwr.obj+strsiz.obj	Elimină fișierul obiect *strlwr.obj* din bibliotecă, în timp ce adaugă fișierul obiect *strsiz.obj*
info_mea.lib +–strlwr.obj	Înlocuiește fișierul obiect *strlwr.obj* în bibliotecă cu fișierul curent pe disc
*info_mea.lib *strupr.obj*	Extrage codul pentru fișierul obiect *strupr.obj* din bibliotecă într-un fișier cu același nume

***Tabelul 694** Operații obișnuite de bibliotecă.*

695 LISTAREA RUTINELOR DINTR-UN FIȘIER BIBLIOTECĂ



Așa cum ați învățat, fișierele bibliotecă dispun de locații convenabile de stocare pentru funcțiile pe care puteți să le utilizați în alte programe. În funcție de compilatorul dumneavoastră, operațiile pe care le suportă programul dumneavoastră de bibliotecă pot diferi. Însă, cele mai multe programe de bibliotecă vă vor permite să vedeți rutinele conținute într-un fișier bibliotecă. De exemplu, utilizând programul de bibliotecă TLIB Borland C++, comanda următoare listează rutinele conținute în fișierul bibliotecă *graphic.lib*:

```
C:\> TLIB \BORLANDC\LIB\GRAPHICS.LIB, CON <ENTER>
```

Pentru a tipări numele funcțiilor conținute în fișierul bibliotecă, înlocuiți în linia de comandă anterioară comanda *CON* cu *PRN*.

696 UTILIZAREA BIBLIOTECILOR PENTRU A REDUCE TIMPUL DE COMPILARE



Pe măsură ce programele dumneavoastră cresc în dimensiune, tot la fel va crește și timpul de compilare a programului dumneavoastră. O modalitate prin care puteți reduce timpul de compilare a programelor dumneavoastră este extragerea funcțiilor de lucru ale programului într-o bibliotecă. În acest fel, când veți compila ulterior programul, nu veți pierde timp recompilând funcțiile. În funcție de numărul de funcții pe care programul dumneavoastră le conține, înlăturarea în acest mod a funcțiilor poate mări semnificativ viteza de compilare a programului. În plus, când scoateți codul funcției din programul dumneavoastră, acesta va deveni mai scurt, mai ușor manevrabil și posibil mai ușor de înțeles.

SĂ ÎNVĂȚĂM MAI MULTE DESPRE CAPACITĂȚILE PROGRAMULUI DE BIBLIOTECĂ

C/C++ 697

Secțiunile despre biblioteci pe care tocmai le-ați citit vă oferă o prezentare a programelor de bibliotecă. Documentația care însoțește compilatorul dumneavoastră descrie caracteristicile programului propriu de bibliotecă în detaliu. De exemplu, programul de bibliotecă TLIB Borland permite invocarea sa fără o linie de comandă pentru afișarea opțiunilor programului:

```
C:\> TLIB <ENTER>
```

De asemenea, programul de bibliotecă LIB Microsoft permite utilizarea următoarei comenzi pentru a afișa opțiunile disponibile ale liniei de comandă:

```
C:\> LIB /? <ENTER>
```

EDITORUL DE LEGĂTURI

C/C++ 698

Așa cum ați învățat, compilatorul convertește fișierul C în limbaj mașină. Dacă programele dumneavoastră apelează funcții pe care le conțin biblioteci sau alte fișiere obiect, editorul de legături va încărca acel cod corespunzător pentru a rezolva apelarea funcțiilor. După rezolvarea tuturor funcțiilor (atât interne, cât și externe) pe care programul le apelează, compilatorul va produce fișierul executabil. Să presupunem, de exemplu, că programul dumneavoastră conține următoarea instrucțiune:

```
void main(void)
  {
    printf("Totul despre C/C++");
  }
```

Atunci când compilați programul, acesta sesizează apelul funcției *printf* din codul programului, dar nu o definește. Ulterior, editorul de legături localizează funcția *printf* în biblioteca run-time, încarcă acel cod în fișierul executabil și apoi actualizează apelarea funcției pentru referința la adresa corectă a funcției în cadrul programului dumneavoastră. Dacă editorul de legături nu este capabil să localizeze funcția, el va afișa un mesaj de eroare pe ecranul dumneavoastră *„unresolved external"* (funcție externă nerezolvată). De aceea, principala funcție a editorului de legături este punerea împreună (legarea) a tuturor părților din codul programului dumneavoastră. Însă, în funcție de editorul dumneavoastră de legături, este posibil să aveți capacitatea de a utiliza editorul de legături pentru a produce o *hartă a legăturilor*, pentru a descrie macheta fișierelor dumneavoastră executabile, pentru a specifica dimensiunea stivei sau a controla utilizarea segmentului de bază al programului. Pentru caracteristicile specifice editorului dumneavoastră de legături, consultați documentația care însoțește compilatorul.

VIZUALIZAREA CAPACITĂȚILOR EDITORULUI DE LEGĂTURI

C/C++ 699

Așa cum ați învățat în secțiunea 698, rolul principal al editorului de legături este de a pune laolaltă toate funcțiile pe care programul le utilizează, într-un fișier executabil. În funcție de

compilatorul dumneavoastră, editorul de legături poate avea și alte capacități. Dacă utilizați *Turbo C++ Lite* sau *Turbo C++*, puteți lista opțiunile liniei de comandă pentru editorul de legături TLINK prin invocarea lui TLINK fără parametri:

```
C:\> TLINK <ENTER>
```

Dacă utilizați Microsoft C/C++, puteți afișa opțiunile liniei de comandă pentru LINK, cum arătăm în continuare:

```
C:\> LINK /? <ENTER>
```

700 UTILIZAREA HĂRȚII DE LEGĂTURI



Așa cum ați învățat, editorul de legături localizează funcțiile externe din cadrul programului dumneavoastră, încarcă funcțiile într-un fișier executabil și actualizează adresele fiecărei referințe la funcții. Atunci când depanați un program, uneori poate că știți că programul are o eroare la o anumită locație de memorie. Dacă utilizați *harta de legături*, care vă arată de unde a încărcat editorul de legături fiecare funcție, puteți să determinați locația erorii. În funcție de editorul dumneavoastră de legături, pașii pe care trebuie să îi parcurgeți pentru a produce o hartă de legături pot diferi. În timp ce *Turbo C++ Lite* nu vă permite crearea unei hărți de legături, puteți utiliza următoarea linie de comandă cu comanda *Turbo C++* Borland pentru a crea fișierul hartă de legături *caut_lng.map*:

```
C:\> BCC -lm CAUT_LNG.C SIR_LNG.OBJ <ENTER>
```

Harta de legături este un fișier ASCII cu un conținut pe care puteți să îl afișați sau să îl tipăriți.

701 UTILIZAREA FIȘIERELOR DE RĂSPUNS PENTRU EDITORUL DE LEGĂTURI



Așa cum ați învățat, în timp ce compilatorul *Turbo C++ Lite* nu acceptă comenzi specifice pentru editorul de legături, majoritatea celorlalte acceptă. Atunci când utilizați comenzile Borland TLINK sau Microsoft LINK, formatul general al comenzii este următorul:

```
TLINK [optiuni] fisiere_obiect, fisier_exe, fisier_harta,
   fisier_biblioteca, fisier_def
LINK [optiuni] fisiere_obiect, fisier_exe, fisier_harta,
   fisier_biblioteca, fisier_def
```

În funcție de numărul de fișiere pe care le legați, liniile dumneavoastră de comandă pot deveni extrem de lungi. Ambele compilatoare acceptă *fișiere de răspuns* astfel că nu trebuie să țineți minte toate numele de fișiere, nici formatul comenzii. Un fișier de răspuns este un fișier ASCII care conține numele de fișiere pe care vreți să le utilizeze editorul de legături pentru fiecare opțiune. Numele de fișiere pentru fiecare tip de fișier trebuie să apară în ordinea din linia de comandă, cu fiecare tip de fișier pe propria sa linie. De exemplu, să analizăm următoarea comandă TLINK:

```
C:\> TLINK CAUT_SIR.OBJ SIR_LNG.OBJ, CAUT_SIR.EXE,
   CAUT_SIR.MAP, OBIBL.LIB
```

În acest caz, fișierul de răspuns va conține următoarele:

```
CAUT_SIR.OBJ SIR_LNG.OBJ
CAUT_SIR.EXE
CAUT_SIR.MAP
OBIBL.LIB
```

Presupunând că numele fișierului de răspuns este *caut_sir.lnk*, puteți invoca TLINK ca mai jos:

```
C:\> TLINK @ CAUT_SIR.LNK <ENTER>
```

Dacă legați mai multe fișiere obiect care nu au loc pe o singură linie, fișierul de răspuns poate continua pe o a doua linie. Pentru a indica o continuare, plasați un semn plus la sfârșitul primei linii.

SIMPLIFICAREA CONSTRUIRII APLICAȚIILOR CU MAKE

C/C++ 702

Pe măsură ce programele dumneavoastră devin mai complexe, ele vor solicita, firește, anumite fișiere antet, module cu cod sursă, fișiere cu cod obiect și biblioteci. Atunci când faceți o modificare programului dumneavoastră, uneori poate că vă va fi dificil să vă amintiți toate fișierele care vor fi afectate de modificare. Pentru a vă simplifica sarcina de reconstruire a unui fișier executabil după ce efectuați modificările în program, multe compilatoare de C vă pun la dispoziție un utilitar MAKE. (Utilitarul MAKE este construit în *Turbo C++ Lite*.)

Utilitarul MAKE este un instrument de programare foarte puternic ce funcționează cu un fișier specific aplicației. Acest fișier (numit adesea ***fișier make***) specifică diferitele fișiere pe care compilatorul le va utiliza pentru a construi o aplicație și listează pașii pe care compilatorul trebuie să îi parcurgă atunci când vă modificați programul. Așa cum ați învățat, fișierele pe care le puneți la dispoziția lui MAKE sunt asemănătoare programelor (adică, ele conțin condiții și instrucțiuni pe care MAKE le evaluează și, eventual, le execută). Există două modalități de utilizare a lui MAKE. În prima, puteți plasa operațiile pe care vreți să le execute MAKE într-un fișier numit MAKEFILE. În cea de a doua, invocați, pur și simplu, MAKE așa cum arătăm în continuare sau încărcați fișierul în *Turbo C++ Lite*.

```
C:\> MAKE <ENTER>
```

Atunci când utilizați un fișier MAKE, MAKE va citi conținutul fișierului și va prelucra corespunzător programul dumneavoastră. Dacă lucrați cu mai multe programe diferite, însă, probabil că veți dori să creați fișiere MAKE care utilizează numele aplicației (a doua metodă). De exemplu, puteți avea un fișier numit *CAUT_LNG.MAK*. Când doriți să invocați MAKE cu un anumit fișier, trebuie să includeți opțiunea *-f*:

```
C:\> MAKE -f CAUT_LNG.MAK <ENTER>
```

Câteva dintre secțiunile următoare descriu operațiile MAKE în detaliu.

Observație: *Atunci când lucrați cu pachete de dezvoltare în C++ bazate pe Windows, cum ar fi Borland C++ 5.02 sau Microsoft Visual C++ 5.0, fișierul* ***proiect*** *creat de pachetul de dezvoltare respectiv, gestionează fișierul MAKE pentru dumneavoastră.*

703 CREAREA UNUI FIȘIER SIMPLU MAKE



MAKE este un instrument care vă ajută să construiți fișiere executabile sau biblioteci după ce ați făcut modificări unui fișier inițial utilizat pentru a construi programul sau biblioteca. Invocați MAKE cu un fișier care conține specificări despre o aplicație, cum ar fi fișierele utilizate pentru a crea aplicația și dependențele fișierelor. Fișierele MAKE sunt conform unui format specific. La început, veți specifica fișierul destinație și fișierele utilizate pentru crearea destinației. De exemplu, să presupunem că doriți să construiți programul *buget.exe* dintr-o sursă numită *buget.c*. În cadrul fișierului MAKE, veți specifica dependențele:

```
BUGET.EXE:    BUGET.C
```

În linia care urmează imediat dependenței, veți include comanda pe care MAKE trebuie să o execute pentru a construi fișierul destinație. În acest caz, cele două linii ale fișierului MAKE vor deveni următoarele:

```
BUGET.EXE:    BUGET.C
   TC BUGET.C
```

Atunci când executați comanda MAKE cu acest fișier MAKE, comanda va examina mai întâi linia dependențelor. Dacă fișierul specificat (în acest exemplu, *buget.exe*) nu există sau dacă fișierul destinație este mai vechi decât oricare alt fișier de care el este dependent (ceea ce înseamnă că ați modificat unul dintre fișierele componente după ce ați compilat ultima dată executabilul), MAKE va executa comanda care urmează. În acest exemplu, MAKE va apela compilatorul (*tc buget.c*), care va recompila programul. Pentru a înțelege mai bine acest proces, creați următorul fișier, *bible.c*:

```
#include <stdio.h>

void main(void)
   {
     printf("Totul despre C/C++");
   }
```

Creați, apoi, fișierul *bible.mak*, care conține următoarele:

```
BIBLE.EXE:    BIBLE.C
   TC BIBLE.C
```

Utilizați opțiunea *-f* pentru a invoca MAKE sau pur și simplu încărcați fișierul *bible.mak* în *Turbo C++ Lite*:

```
C:\> MAKE -f BIBLE.MAK <ENTER>
```

Deoarece fișierul *bible.exe* nu există, MAKE va executa comanda de construire a acestuia. După ce MAKE își încheie execuția, invocați MAKE a doua oară utilizând aceeași comandă. Pentru că acum fișierul *bible.exe* deja există și pentru că fișierul este mai nou decât fișierul *bible.c*, MAKE nu va executa comanda. Editați fișierul *bible.c* și modificați instrucțiunea *printf* într-un mod oarecare (de exemplu, înlocuiți ieșirea cu "Acesta este un test"). Repetați comanda MAKE. Pentru că fișierul *bible.c* este mai vechi decât fișierul *bible.exe* în cadrul acestei execuții, MAKE va construi un nou fișier EXE.

UTILIZAREA FIȘIERELOR CU DEPENDENȚE MULTIPLE CU MAKE

C/C++ 704

După cum ați învățat, MAKE este un instrument care vă ajută să construiți aplicații după una sau mai multe modificări ale fișierelor componente ale aplicației. Atunci când utilizați MAKE, uneori fișierul destinație este dependent de mai multe fișiere. De exemplu, să presupunem că programul *buget.exe* este dependent de fișierul C *buget.c*, de fișierul antet *buget.h* și de fișierul bibliotecă *buget.lib*. În cadrul lui MAKE, puteți specifica dependențele, ca mai jos:

```
BUGET.EXE:      BUGET.C BUGET.H BUGET.LIB
   TC BUGET.C BUGET.LIB
```

COMENTAREA FIȘIERELOR MAKE

C/C++ 705

Toate fișierele MAKE pe care le-a prezentat această carte sunt de dimensiuni reduse și relativ explicite. Pe măsură ce complexitatea fișierelor dumneavoastră MAKE crește, puteți să adăugați comentarii care să explice prelucrările efectuate de aceste fișiere. Pentru a plasa un comentariu într-un fișier MAKE, pur și simplu puneți semnul diez (#) oriunde în fișier. MAKE va considera orice text care urmează pe linia curentă, drept comentariu. Următorul fișier MAKE ilustrează modul de utilizare a comentariilor:

```
# Construieste programul de gestionare a bugetului BUGET.EXE
# Fisier MAKE creat la data de: 10.12.97 de Kris Jamsa si
  Lars Klander
BUGET.EXE:      BUGET.C BUGET.H BUGET.LIB
   TC BUGET.C BUGET.LIB     # BUGET.LIB contine functii de
     contabilitate generala
```

LINIILE DE COMANDĂ ȘI MAKE

C/C++ 706

După cum ați învățat, dacă un fișier destinație este mai vechi decât un fișier pe baza căruia ați creat fișierul destinație, MAKE va executa o comandă specifică. În exemplele pe care le-ați văzut până acum, MAKE apela comanda TC pentru a compila și a lega fișierele corespunzătoare. MAKE poate efectua orice comandă și, mai mult, acceptă fără probleme operatorii de redirectare din DOS: *input*(<), *output* (>) și *append* (>>). Următoarea comandă, de exemplu, indică lui MAKE să compileze un anumit program, redirectând ieșirea compilatorului către imprimantă:

```
BUGET.EXE:      BUGET.C BUGET.H BUGET.LIB
   TC BUGET.C BUGET.LIB > PRN
```

În plus față de acceptarea operatorilor de redirectare ai sistemului DOS, MAKE acceptă și doi operatori speciali, << și &&. Semnul dublu de „mai mic" (<<) indică lui MAKE să redirecteze sursa standard de intrare a comenzii. Însă, în loc să redirecteze sursa de intrare către fișier, MAKE utilizează textul care urmează imediat până la un delimitator pe care îl specificați, ca intrare redirecționată. De exemplu, următorul fișier MAKE cere calculatorului să redirecteze textul "Totul despre C/C++" către comanda SHOWMSG:

```
UNFISIER.EXE:    UNFISIER.C
   SHOWMSG << ^Totul despre C/C++
^
```

În acest caz, comanda utilizează semnul ^ ca delimitator de intrare. MAKE vă permite să utilizaţi orice caracter, exceptând caracterul diez (#) sau backslash (\) ca delimitator. Prima linie care începe cu delimitatorul pe care l-aţi specificat, marchează sfârşitul textului pe care doriţi să îl redirectaţi către comanda SHOWMSG. Operatorul && este similar, dar el nu realizează redirectarea. În schimb, operatorul && creează un fişier temporar care conţine textul care apare între delimitatorii specificaţi. În timpul executării comenzii, MAKE înlocuieşte operatorul cu numele fişierului temporar pe care îl înlocuieşte. Veţi utiliza operatorul && cel mai frecvent pentru a crea un fişier de răspuns la editarea legăturilor, ca mai jos:

```
UNFISIER.EXE:    UNFISIER.C
   TLINK && ^
   UNFISIER.C
   UNFISIER.EXE
   UNFISIER.MAP
   UNFISIER.LIB
^
```

Ca în exemplul precedent, prima linie care începe cu delimitatorul specificat marchează sfârşitul fişierului temporar.

Observaţie: *Nu puteţi crea fişiere MAKE care manipulează direct editorul de legături în* ***Turbo C++ Lite.***

707 PLASAREA DEPENDENŢELOR MULTIPLE ÎNTR-UN FIŞIER MAKE

C/C++

Când construiţi un sistem vast de programe, puteţi să aveţi, de fapt, câteva fişiere executabile diferite. Decât să manevraţi câteva fişiere MAKE diferite, puteţi să creaţi un singur fişier care să cuprindă dependenţele corelate pentru întregul sistem de programe. De exemplu, următorul fişier MAKE conţine regulile de care aveţi nevoie pentru a construi programele *buget.exe*, *plati.exe* şi *taxe.exe*:

```
BUGET.EXE:     BUGET.C BUGET.H
   TC BUGET.C

PLATI.EXE:     PLATI.C PLATI.H
   TC PLATI.C

TAXE.EXE:     TAXE.C TAXE.H
   TC TAXE.C
```

Atunci când executaţi MAKE cu fişierele arătate anterior, el va începe cu prima intrare din fişier. În acest exemplu, dacă MAKE detectează că compilatorul trebuie să reconstruiască primul fişier destinaţie, MAKE va executa comanda corespunzătoare. Apoi MAKE va continua executarea comenzilor corespunzătoare celorlalte fişiere pe care le specifică exemplul.

REGULI EXPLICITE ȘI IMPLICITE ALE COMENZII MAKE

C/C++ 708

Atunci când creați un fișier MAKE, intrările pe care le plasați în fișierul care îi indică dependențele fișierelor și operațiile corespunzătoare sunt denumite *reguli*. MAKE acceptă reguli *implicite și explicite*. O regulă explicită este o regulă care definește unul sau mai multe nume de destinație, zero sau mai multe fișiere dependente și zero sau mai multe comenzi. Numele de fișiere din cadrul regulilor explicite pot fi numele unei căi complete în DOS sau caractere de înlocuire. Toate exemplele de fișiere MAKE prezentate până acum în această carte au utilizat reguli explicite. Regulile implicite, pe de altă parte, sunt mult mai generale. O regulă implicită corespunde tuturor fișierelor cu anumită extensie. MAKE utilizează o regulă implicită atunci când nu prevedeți o regulă explicită într-un fișier destinație. De exemplu, trebuie să specificați că un fișier OBJ depinde de un fișier C. Următoarea regulă implicită indică lui MAKE să compileze toate fișierele C ale căror fișiere în cod sursă sunt mai noi decât fișierele OBJ corespunzătoare:

```
.C.OBJ:
  TC $<
```

Sintaxa pentru o regulă implicită este tipul de fișier dependent (C) urmat de tipul fișierului destinație (OBJ). Regula utilizează o macrocomandă specială ($<) care, așa cum veți învăța, cere comenzii MAKE să utilizeze numele complet al fișierelor C corespunzătoare. Plasarea unei singure reguli implicite în cadrul unui fișier MAKE nu are efect atunci când MAKE rulează – cu alte cuvinte, un fișier MAKE trebuie să includă cel puțin o regulă explicită. MAKE va utiliza regula implicită numai atunci când întâlnește un fișier destinație pentru care nu ați prevăzut o regulă explicită.

UTILIZAREA MACROCOMENZILOR MAKE

C/C++ 709

O macrocomandă *MAKE* este un simbol pe care MAKE îl înlocuiește cu o anumită valoare. Puteți utiliza o macrocomandă în MAKE în multe scopuri. De exemplu, următoarea macrocomandă, *MEM_MODEL*, definește opțiunile pe care compilatorul le necesită pentru a selecta modelul *small* de memorie:

```
MEM_MODEL = -ms
```

Pentru a utiliza valoarea unei macrocomenzi în cadrul fișierului MAKE, plasați numele de macro între paranteze care sunt precedate de semnul $. De exemplu, următoarea linie de comandă utilizează macrocomanda *MEM_MODEL*:

```
NUMEFIS.EXE:    NUMEFIS.C
  BCC $(MEM_MODEL) NUMEFIS.C
```

Observație: *Din cauza modului în care* ***Turbo C++ Lite*** *implementează controlul modelului de memorie, această macrocomandă nu va funcționa în* ***Turbo C++ Lite****, de aceea apare comanda de invocare din* ***Turbo C++****.*

710 MACROCOMENZI MAKE PREDEFINITE



După cum ați învățat, o macrocomandă MAKE este un simbol pe care MAKE îl înlocuiește cu o anumită valoare. MAKE dispune de câteva macrocomenzi predefinite pe care puteți să le utilizați în cadrul fișierelor MAKE. După cum macrocomenzile sunt utilizate în cadrul regulilor explicite sau implicite, valoarea pe care MAKE o substituie pentru simbol va diferi. Tabelul 710.1 prezintă modul în care sunt utilizate macrocomenzile MAKE predefinite în regulile explicite. De asemenea, tabelul 710.2 prezintă modul de utilizare a macrocomenzilor în regulile implicite.

Numele macrocomenzii	Valoarea returnată
$*	Numele de bază al fișierului dependent cu cale
$&	Numele de bază al fișierului dependent fără cale
$.	Numele complet al fișierului dependent fără cale
$**	Numele complet al fișierului dependent cu cale
$<	Numele complet al fișierului dependent cu cale
$?	Numele complet al fișierului dependent cu cale

Tabelul 710.1 *Macrocomenzile MAKE predefinite pentru regulile explicite.*

Numele macrocomenzii	Valoarea returnată
$*	Numele de bază al fișierului destinație cu cale
$&	Numele de bază al fișierului destinație fără cale
$.	Numele complet al fișierului destinație fără cale
$**	Toate numele fișierelor dependente
$<	Numele complet al fișierului destinație cu cale
$?	Toate fișierele dependente neactuale

Tabelul 710.2 *Macrocomenzile MAKE predefinite pentru regulile implicite.*

Următorul fișier MAKE, de exemplu, creează o regulă implicită care îi indică relațiile dintre fișierele cu extensiile OBJ și C:

```
.C.OBJ:
   TC $<
```

În acest exemplu, MAKE va utiliza regula implicită pentru a expanda macrocomanda $< în numele fișierului destinație și cale.

711 PROCESAREA CONDIȚIONATĂ CU MAKE



În capitolul despre macrodefiniții al acestei cărți, ați învățat cum se utilizează directive către preprocesor cum ar fi *#if, #elif, #else* și *#endif.* În mod similar, MAKE prevede instrucțiuni de procesare condiționată care încep cu un semn de exclamare (!) cum ar fi *!if, !else, !elif* și *!endif.* De asemenea, puteți utiliza directivele *!ifdef, !ifndef* și *!undef,* pentru a testa macrocomenzile definite sau pentru a elimina definiția unei macrocomenzi. Dacă o directivă condițională se evaluează ca adevărată, MAKE va executa regulile care urmează. Dacă

directiva este falsă, MAKE nu va prelucra regulile corespunzătoare. Următoarele instrucțiuni ilustrează câteva instrucțiuni condiționale:

```
!ifdef nume_macro
    # Testeaza daca macrocomanda nume_macro este definita
    # instructiuni
!endif
!if $(Valoare) > 5
    # Testeaza daca valoarea macrocomenzii Valoare este > 5
    # instructiuni
!endif
!if ! $d(nume_macro)
    # Testeaza daca macrocomanda nume_macro nu este definita
    # instructiuni
!endif
```

TESTAREA UNEI MACROCOMENZI MAKE

C/C++712

După cum ați învățat, MAKE vă permite să definiți propriile dumneavoastră macrocomenzi. În funcțiile de prelucrările pe care fișierele dumneavoastră MAKE le execută, puteți să testați dacă o anumită macrocomandă este definită. Pentru aceasta, puteți să utilizați testul *$d(macro)*. Dacă macrocomanda este definită, testul va returna valoarea 1. Dacă macrocomanda nu este definită, rezultatul va fi 0. Următoarea instrucțiune utilizează operatorul condițional *!if* al lui MAKE pentru a determina dacă macrocomanda *MEM_MODEL* este definită. Dacă macrocomanda nu este definită, instrucțiunile îi vor atribui valoarea modelului de memorie *small*, după cum arătăm mai jos:

```
!if ! $d(MEM_MODEL)
MEM_MODEL = -ms
!endif
```

În plus față de utilizarea testului *$d(macro)*, macrocomenzile dumneavoastră pot efectua o prelucrare echivalentă, utilizând instrucțiunile condiționale *!ifdef* și *!ifndef*. Dacă ulterior veți dori să eliminați definiția unei macrocomenzi, puteți utiliza instrucțiunea *!undef* pentru a face aceasta, după cum arătăm mai jos:

```
!undef nume_macro
```

***NOTĂ:** După cum ați învățat, această atribuire deosebită nu se va efectua într-un fișier MAKE din **Turbo C++ Lite**, din cauza modului în care acesta gestionează modelele de memorie.*

INCLUDEREA UNUI AL DOILEA FIȘIER MAKE

C/C++713

Dacă fișierele dumneavoastră MAKE au în general aceeași formă, atunci puteți găsi convenabil să plasați regulile dumneavoastră implicite frecvent utilizate într-un fișier MAKE denumit *implicit.mak*. La începutul fiecărui fișier MAKE, puteți include fișierul utilizând directiva *!include*, ca mai jos:

```
!include "IMPLICIT.MAK"
```

714 UTILIZAREA MODIFICATORILOR DE MACROCOMENZI



După cum ați învățat în secțiunea 710, MAKE predefinește câteva macrocomenzi diferite pe care fișierele dumneavoastră MAKE le pot utiliza pentru a obține fișierul destinație sau cel dependent. Pentru ca fișierele MAKE să aibă un control mai bun asupra numelor de fișiere pe care aceste macrocomenzi le returnează, MAKE vă permite să folosiți modificatorii B, D, F și R, detaliați în tabelul 714.

Modificator	Scop
$(macroB)	Returnează numai numele de bază
$(macroD)	Returnează unitatea de disc și directorul
$(macroF)	Returnează numele de bază și extensia
$(macroR)	Returnează unitatea de disc, directorul și numele de bază

Tabelul 714 *Modificatorii pentru macrocomenzile MAKE*

Următoarea instrucțiune, de exemplu, utilizează modificatorul D cu macrocomanda *$<* pentru a copia fișiere din directorul fișierului destinație într-un director backup:

```
C:\SUBDIR\CAPITOLE.EXE:     CAPITOLE.C
   COPY $(<D)*.C        C:\BACKUP
   TC CAPITOLE.C
```

NOTĂ: *Acest fișier MAKE execută două comenzi în cadrul regulii. Prima comandă copiază fișierele cu extensia C într-un director denumit BACKUP, iar a doua comandă compilează fișierul sursă.*

715 ÎNCHEIEREA CU O EROARE A UNUI FIȘIER MAKE



În funcție de prelucrarea pe care o efectuează fișierul dumneavoastră MAKE, e posibil ca MAKE să își încheie prelucrarea și să afișeze un mesaj de eroare către utilizator. În aceste cazuri, puteți utiliza directive *!error*. Următoarele instrucțiuni, de exemplu, testează pentru a vedea dacă macrocomanda *MEM_MODEL* este nedefinită. Dacă macrocomanda este nedefinită, MAKE va afișa un mesaj de eroare către utilizator și va încheia prelucrarea:

```
!ifndef MEM_MODEL
!error Ending program build-define the macro MEM_MODEL
!endif
```

716 DEZACTIVAREA AFIȘĂRII UNEI COMENZI



În mod implicit, MAKE va afișa fiecare comandă înainte de executarea sa. Pentru a dezactiva afișarea comenzii, precedați numele comenzii cu un simbol @. De exemplu, următoarea instrucțiune utilizează simbolul @ pentru a dezactiva afișarea comenzii TC:

```
CAPITOLE.EXE: CAPITOLE.C
   @TC CAPITOLE.C
```

Dacă doriți să dezactivați și ieșirea comenzii, pur și simplu redirectați ieșirea către un dispozitiv NUL:

```
CAPITOLE.EXE: CAPITOLE.C
   @TC CAPITOLE.C > NUL
```

UTILIZAREA FIȘIERULUI BUILTINS.MAK

C/C++717

După cum ați învățat, există multe reguli implicite pe care le puteți folosi în mod regulat. O modalitate pentru a vă asigura că toate fișierele dumneavoastră MAKE pot utiliza regulile implicite este plasarea regulilor uzuale într-un fișier special, numit *builtins.mak*. De fiecare dată când apelați MAKE, acesta va căuta fișierul *builtins.mak*. Dacă fișierul există, MAKE va procesa imediat informațiile conținute de acesta. Dacă fișierul nu există, MAKE va continua prelucrarea utilizând MAKEFILE sau fișierul pe care l-ați specificat în linia de comandă. Următorul fișier *builtins.mak* conține regula implicită pentru conversia fișierelor C sau OBJ:

```
.C.OBJ:
   TC $<
```

EFECTUAREA PRELUCRĂRII STĂRII DE IEȘIRE ÎN MAKE

C/C++718

Atunci când MAKE execută o comandă, e posibil ca MAKE să evalueze valoarea de stare de ieșire a comenzii și apoi, fie să continue, fie să se încheie. Dacă precedați numele comenzii cu o cratimă urmată de o valoare, MAKE va compara valoarea de stare de ieșire cu valoarea pe care fișierul MAKE o specifică. Dacă starea de ieșire este *mai mare decât* valoarea, MAKE va renunța la construirea programului curent. De exemplu, următoarea instrucțiune compară starea de ieșire a comenzii *showfile* cu 3. Dacă starea de ieșire este mai mare decât 3, MAKE va renunța la construire, cum arătăm mai jos:

```
TEST.EXE: CAPITOLE.C
   -3 SHOWFILE $?
```

Dacă doriți ca MAKE să ignore starea de ieșire a comenzii, precedați numele comenzii cu o cratimă fără o valoare corespunzătoare:

```
   -CC TEST.C
```

APELAREA ȘI MODIFICAREA SIMULTANĂ A UNEI MACROCOMENZI

C/C++719

Așa cum ați învățat, MAKE vă permite să definiți propriile dumneavoastră macrocomenzi. În funcție de prelucrarea efectuată de fișierul MAKE, puteți să modificați și să utilizați imediat o macrocomandă. De exemplu, să presupunem că definiți macrocomanda *INPUT_FIS* cum arătăm în continuare:

```
INPUT_FIS = BUGET.C
```

Puteți să utilizați macrocomanda cum arătăm în continuare:

```
BUGET.EXE:     $(INPUT_FIS)
   TC $(INPUT_FIS)
```

Apoi, presupunem că doriți să copiați fișierul de intrare într-un alt fișier cu același nume, dar cu extensia SAV. Puteți să modificați macrocomanda, înlocuind .C cu .SAV și apoi să utilizați imediat noua definiție. Următoarea comandă ilustrează cum se face copia:

```
COPY $(INPUT_FIS)    $(INPUT_FIS:.C=.SAV)
```

Prima parte a comenzii COPY utilizează numele de fișier *buget.c*. Cea de a doua parte a comenzii înlocuiește .C cu .SAV pentru a crea numele de fișier *buget.sav*.

720 EXECUTAREA COMENZII MAKE PENTRU MAI MULTE FIȘIERE DEPENDENTE



După cum ați învățat, uneori este posibil ca un fișier destinație să fie dependent de două sau mai multe fișiere. În funcție de prelucrarea efectuată de fișierul MAKE, e posibil ca MAKE să execute o comandă anumită pentru fiecare fișier. Pentru a face aceasta, precedați, pur și simplu, numele comenzii cu semnul ampersand (&). Următoarea regulă, de exemplu, indică lui MAKE să compileze în mod individual fiecare fișier dependent neactual:

```
BUGET.EXE:      BUGET.C CONTURI.C PLATI.C
   & TC $?
```

721 DETERMINAREA PREZENȚEI COPROCESORULUI MATEMATIC



Dacă programele dumneavoastră efectuează operații matematice complexe, uneori este posibil să utilizați coprocesorul matematic al calculatorului pentru a mări performanțele programelor dumneavoastră. Pentru a ajuta programele dumneavoastră să beneficieze de avantajele coprocesorului matematic, există câteva biblioteci de la terțe firme care dispun de funcții utilizate în mod obișnuit. Înainte însă ca programul dumneavoastră să utilizeze astfel de funcții, programul ar trebui să verifice dacă este prezent coprocesorul matematic. Pentru asemenea cazuri, multe compilatoare de C definesc variabila globală *_8087*, care conține valoarea 1 dacă este prezent coprocesorul și 0 dacă acesta nu este prezent. Următorul program, *test_mat.c*, ilustrează modul de utilizare a variabilei globale *_8087*:

```
#include <stdio.h>
#include <dos.h>

void main(void)
   {
     if (_8087)
       printf("A gasit coprocesorul matematic\n");
     else
       printf("Nu a gasit coprocesorul matematic\n");
   }
```

Puteți utiliza intrarea de mediu *87* pentru a controla valoarea pe care compilatorul de C o atribuie variabilei *_8087*. Pentru a stabili valoarea variabilei la 1, atribuiți intrării *87* valoarea *Yes*, așa cum arătăm în continuare:

```
C:\> SET 87=Yes <ENTER>
```

De asemenea, pentru a stabili variabila la 0, atribuiți intrării de mediu valoarea *No*.

FIȘIERUL CTYPE.H ȘI MACROCOMENZILE ISTYPE

C/C++ 722

Capitolul despre macrodefiniții al acestei cărți prezintă câteva macrocomenzi care testează dacă un caracter este majusculă, minusculă, alfanumeric și așa mai departe. Dacă examinați fișierul antet *ctype.h*, veți găsi definiții de macrocomenzi similare cu următoarele:

```
#define isalpha(c)   (_ctype[(c) + 1] & (_IS_UPP | IS_LOW))
#define isascii(c)   ((unsigned)(c) < 128)
#define iscntrl(c)   (_ctype[(c) + 1] & _IS_CTL)
#define isdigit(c)   (_ctype[(c) + 1] & _IS_DIG)
#define isgraph(c)   ((c) >= 0x21 && (c) <= 0x7e)
#define islower(c)   (_ctype[(c) + 1] & _IS_LOW)
```

Pentru a reduce timpul de procesare pe care testarea macrocomenzilor îl cere, multe compilatoare de C definesc o variabilă globală numită *ctype*, care conține valorile care definesc fiecare caracter ASCII. Utilizând aceste valori, macrocomenzile *istype* pot utiliza mai rapid operațiile pe biți pentru a executa testările necesare. Următorul program, *ctype.c*, afișează valorile pe care compilatorul le utilizează pentru fiecare caracter ASCII:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
  {
    int caract_ascii;
    for (caract_ascii = 0; caract_ascii < 128; caract_ascii++)
      if (isprint(caract_ascii))
        printf("Valoarea ASCII %d varianta (hex) %x ASCII %c\n",
               caract_ascii, _ctype[caract_ascii], caract_ascii);
      else
        printf("Valoarea ASCII %d varianta (hex) %x ASCII %c\n",
               caract_ascii, _ctype[caract_ascii], caract_ascii);
  }
```

CONTROLUL VIDEO DIRECT

C/C++ 723

Fișierul antet *conio.h* definește prototipurile pentru funcțiile care execută operații I/O de la consolă, cum ar fi *cputs*. Pentru a crește performanța acestor funcții de I/O de la consolă, majoritatea compilatoarelor de C ocolesc serviciile DOS și BIOS și scriu ieșirea direct în memoria video a PC-ului. Deși majoritatea operațiilor video sunt standard de la un PC la altul, puteți întâlni o placă video care nu acceptă operații video directe. Dacă apar astfel de

erori, puteți utiliza variabila globală *directvideo* pentru a controla dacă PC-ul utilizează rutinele video BIOS pentru a executa ieșirea programului sau dacă programul dumneavoastră execută operații I/O directe. Dacă stabiliți valoarea variabilei la 1, rutinele vor executa ieșiri video directe. Dacă valoarea este 0, rutinele vor executa ieșirile utilizând serviciile BIOS.

724 DETECTAREA ERORILOR DE SISTEM ȘI MATEMATICE



Câteva dintre funcțiile de bibliotecă run-time de C pe care le prezintă această carte, atribuie valori variabilei globale *errno* când apare o eroare. Atunci când programele dumneavoastră utilizează aceste funcții, ar trebui să testați atât valoarea returnată a funcției, cât și valoarea variabilei *errno*. Tabelul 724 definește constantele pe care funcțiile le atribuie variabilei globale *errno*.

Constanta	Semnificația
E2BIG	Lista de argumente este prea lungă
EACCES	Permisiune refuzată
EBADF	Indicator de fișier greșit
ECONTRL	Eroare în blocurile de control ale memoriei
ECURDIR	Încercare de eliminare a directorului curent
EDOM	Un argument violează domeniul de valori acceptate
EEXIST	Fișierul există deja
EFAULT	Eroare necunoscută
EINVACC	Specificator de acces nevalid
EINVAL	Valoare argument nevalidă
EINVDAT	Date argument nevalide
EINVDRV	Specificator de unitate nevalid
EINVENV	Mediu nevalid
EINVFMT	Format de argument nevalid
EINVFNC	Număr de funcție nevalid
EINVMEM	Specificator de bloc de memorie nevalid
ENFILE	Prea multe fișiere deschise
ENMFILE	Nu mai sunt fișiere
ENODEV	Nu există un astfel de dispozitiv
ENOENT	Intrare nevalidă (fișier sau director)
ENOEXEC	Eroare de format la EXEC
ENOFILE	Nu există un astfel de fișier sau director
ENOMEM	Memorie insuficientă
ENOPATH	Calea nu e găsită

Constanta	Semnificația
ENOTSAM	Nu este același dispozitiv
ERANGE	Rezultatul funcției este în afara intervalului de valori valide
EXDEV	Dispozitiv suprapus
EZERO	Eroare zero

Tabelul 724 *Valorile constante pe care funcțiile le atribuie variabilei globale* ***errno.***

Observație: *Compact discul care însoțește această carte include fișierul text* ***win_math.txt****, care listează erorile matematice din mediul Windows.*

AFIȘAREA MESAJELOR DE EROARE PREDEFINITE

C/C++ 725

În secțiunea 724 ați învățat că diferite funcții matematice și de sistem atribuie variabilei globale *errno* valori de stare specifice pe care programele dumneavoastră le pot citi pentru a obține informații despre cauza unei erori. În funcție de prelucrările programului dumneavoastră, puteți să afișați un mesaj predefinit atunci când apare o eroare. Pentru a ajuta programul dumneavoastră să proceseze erorile, compilatorul de C dispune de o variabilă globală numită *sys_erlist* care conține șirul de caractere al mesajelor de eroare pentru majoritatea erorilor. În plus, pentru a mări portabilitatea programelor dumneavoastră, matricea conține mesaje de eroare din mediul Unix.

Compilatorul atribuie, de asemenea, variabilei globale *sys_errlist* numărul mesajelor de eroare din tablou. Următorul program, *er_msg.c*, utilizează matricea *sys_errlist* pentru a afișa mesajele de eroare predefinite:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
  {
    int eroare;
    for (eroare = 0; eroare < sys_nerr; eroare++)
      printf("Eroare %d %s\n", eroare, sys_errlist[eroare]);
  }
```

DETERMINAREA NUMĂRULUI VERSIUNII SISTEMULUI DE OPERARE

C/C++ 726

Dacă dezvoltați aplicații pentru mediu DOS, uneori poate că programele dumneavoastră vor trebui să cunoască numărul versiunii sistemului de operare. În aceste cazuri, programele dumneavoastră pot utiliza variabilele globale predefinite *_osmajor* și *_osminor*, care conțin numărul mai mare și, respectiv, mai mic al versiunii sistemului de operare. În plus, unele compilatoare dispun de variabila *_version*. Octetul inferior al constantei conține numărul mai mare al versiunii și octetul superior conține numărul mai mic al versiunii. În cazul lui DOS 6.0, de exemplu, variabila *_osmajor* va conține valoarea 6, iar variabila *_osminor* va fi 0.

Următorul program, *os_ver.c*, utilizează variabilele globale de versiune pentru a afișa numărul de versiune al sistemului de operare:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    printf("Numarul versiunii sistemului de operare %d.%d\n",
           _osmajor, _osminor);
    printf("Numarul versiunii sistemului de operare %d.%d\n",
           _version & 255, _version >> 8);
  }
```

Observație: *Dacă programul anterior nu este compilat în sistemul dumneavoastră, comentați instrucțiunea care utilizează variabila* ***_version*** *și includeți fișierul antet* ***stdlib.h.***

727 PORTABILITATEA



Portabilitatea este o măsură a ușurinței cu care puteți să mutați programul dumneavoastră de la un sistem la altul. De exemplu, atunci când scrieți un program utilizând limbajul de asamblare pentru PC, acesta este foarte dificil de mutat pe o stație de lucru care utilizează un alt limbaj de asamblare. Însă, dacă ați scris același program în C, aveți nevoie să faceți numai câteva mici modificări programului înainte de a-l compila și executa în alt sistem. Atunci când programați, ar trebui să aveți în vedere portabilitatea. Deseori, puteți să utilizați același cod, scris pentru un anumit program, pentru multe alte programe și puteți economisi astfel mult timp de programare și de testare dacă vă concentrați pe scrierea unui cod portabil. Pentru a crește portabilitatea programelor dumneavoastră, să aveți în vedere următoarele, atunci când programați:

- Evitați serviciile sistemului de operare ori de câte ori este posibil. Bazați-vă pe rutinele de bibliotecă run-time de C.
- Evitați funcțiile și variabilele globale care sunt specifice compilatorului dumneavoastră. În cele mai multe cazuri, compilatorul va preceda numele acestor funcții și variabile globale cu liniuță de subliniere (_), ca de exemplu *_8087*.
- Nu faceți presupuneri în legătură cu dimensiunea cuvântului pentru o mașină. De exemplu, pe un PC, o variabilă de tip *int* are în mod obișnuit 16 biți, dar pe alte mașini poate să aibă 32 biți.
- Nu accesați locații specifice de hardware sau nu vă bazați pe întreruperi specifice dacă performanța programului dumneavoastră nu cere neapărat aceasta.
- Încercați întotdeauna să corectați și să eliminați mesajele de avertizare ale compilatorului.
- Nu faceți presupuneri asupra modelului de memorie care poate că nu există într-un mediu Unix.
- Restricționați la cât mai puțin posibil funcțiile utilizate în codul dumneavoastră care depind de hardware sau de sistemul de operare.

EXECUTAREA INSTRUCȚIUNII GOTO NELOCALĂ

C/C++728

În primul capitol al acestei cărți, ați învățat că instrucțiunea *goto* permite ca execuția programelor dumneavoastră să se ramifice de la o locație la alta. După cum ați învățat, eticheta destinație la care doriți ca *goto* să sară trebuie să se situeze în funcția curentă. În funcție de programele dumneavoastră, uneori va trebui să ramificați către o destinație din afara funcției curente (numită *goto nelocală*). Pentru a executa o instrucțiune *goto nelocală*, programele dumneavoastră pot utiliza funcțiile *setjmp* și *longjmp*:

```
#include <setjmp.h>

void longjmp(jmp_buf locatie, int val_return);
void setjmp(jmp_buf locatie);
```

Pentru început, programul dumneavoastră va utiliza *setjmp* pentru a stoca locația curentă (cunoscută, de asemenea, ca *task state*) în bufferul *locatie*. După aceea, programele dumneavoastră pot face un salt la acea locație utilizând *longjmp*. Când programele dumneavoastră invocă pentru prima oară funcția *setjmp*, funcția va returna 0. Apoi, când programul apelează funcția *longjmp*, aceasta se va returna la locația anterior păstrată de funcția *setjmp* și va rezulta valoarea returnată specificată în cadrul parametrului *val_return* al funcției *longjmp*. Următorul program, *longjmp.c*, ilustrează instrucțiunea *goto nelocală*:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf locatie; // Variabila globala
void functie(void)
  {
   printf("Pe punctul de a incepe longjmp\n");
   longjmp(locatie, 1); // Returneaza 1
  }
void main(void)
  {
   if (setjmp(locatie) != 0) // Salveaza locatia curenta
     {
      printf("Se returneaza de la longjmp\n");
      exit(1);
     }
   functie();
  }
```

OBȚINEREA IDENTIFICATORULUI DE PROCES (PID)

C/C++729

Într-un mediu multitasking, sistemul de operare atribuie fiecărui program un identificator unic (ID), numit PID. Sistemul de operare Unix dispune de funcția numită *getpid* care returnează valoarea PID a programului. Multe compilatoare de mediu DOS dispun de o funcție similară *getpid*:

```
#include <process.h>
unsigned getpid(void);
```

În cadrul sistemului DOS, PID este în realitate o adresă de segment a segmentului prefix al programului (PSP). Dacă aveți două sau mai multe programe active rezidente în memorie, fiecare program va avea un unic PID, pentru că fiecare are o unică adresă de segment PSP. Următorul program, *getpid.c*, afișează valoarea PID a programului:

```
#include <stdio.h>
#include <process.h>

void main(void)
  {
    printf("Proces id: %X\n", getpid());
  }
```

Observație: *Windows gestionează procesele și corespondentele lor, firele de execuție, puțin diferit decât prin PID ca în sistemul DOS. Secțiunile de la 1376 la 1400 abordează gestionarea proceselor și a firelor de execuție în detaliu.*

730 INVOCAREA UNEI COMENZI INTERNE DOS 

Câteva dintre secțiunile acestui capitol v-au prezentat modalități prin care programele dumneavoastră pot invoca fișiere executabile (EXE și COM). În funcție de programul dumneavoastră, puteți să invocați o comandă DOS internă sau un fișier de comenzi. În astfel de cazuri, programele dumneavoastră pot utiliza funcția *system*:

```
#include <stdlib.h>
int system(const char *comanda);
```

Parametrul *comanda* este un șir de caractere care conține numele comenzii interne sau externe DOS sau al fișierului de comenzi. Dacă funcția *system* reușește să execute comanda, ea va returna valoarea 0. Dacă apare o eroare, funcția *system* va returna valoarea –1 și va atribui variabilei globale *errno* una dintre valorile listate în tabelul 730.

Valoarea	Semnificație
ENOENT	Nu există un astfel de fișier
ENOMEM	Nu este suficientă memorie
E2BIG	Lista de argumente prea lungă
ENOEXEC	Eroare în formatul *exec*

Tabelul 730 *Valorile* ***errno*** *pe care le returnează funcția* ***system.***

Funcția *system* generează o copie a fișierului *command.com* pentru a executa comanda specificată. Funcția utilizează intrarea de mediu COMSPEC pentru a localiza procesorul comenzii. Următorul program, *system.c*, utilizează funcția *system* și comanda DOS *dir* pentru a afișa lista directoarelor:

```
#include <stdlib.h>
void main(void)
```

```
{
   if (system("DIR"))
     printf("Eroare la invocarea DIR\n");
}
```

UTILIZAREA VARIABILEI GLOBALE _PSP

C/C++ 731

De fiecare dată când executați un program, sistemul DOS încarcă programul în memorie imediat după un buffer de 256 de octeți, numit segmentul prefix al programului (PSP – *program segment prefix*). Segmentul prefix al programului conține informații despre linia de comandă, un pointer la copia de mediu a programului, informații despre tabelele de fișiere și așa mai departe. Figura 731 ilustrează conținutul segmentului prefix al programului.

Offset	Conținut
0H	Instrucțiunea Int 20H
2H	Vârful adresei de segment a memoriei
4H	Rezervat
5H	Apelare *far* a dispecerului DOS
AH	Vectorul Int 22H
EH	Vectorul Int 23H
12H	Vectorul Int 24H
16H	Rezervat
2CH	Adresa de segment a copiei de mediu
2EH	Rezervt
5CH	FCB 1 implicit
6CH	FCB 2 implicit
7CH	Rezervat
80H	Lungimea în octeți aliniei de comandă
81H	Linia de comandă
FFH	

Figura 731 *Conținutul segmentului prefix al programului.*

Pentru că ați început să faceți ca programele dumneavoastră să lucreze cu comenzi interne DOS, este posibil ca uneori programele dumneavoastră să acceseze informația pe care o conține PSP. De aceea, sistemul DOS dispune de un serviciu de sistem care returnează adresa PSP. Pentru a simplifica aceste programe, unele compilatoare de C definesc variabila globală *_psp*, care conține adresa de segment PSP. Următorul program, *psp_adr.c*, utilizează variabila globală *_psp* pentru a afișa adresa PSP:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    printf("Segmentul prefix al programului incepe la %X\n", _psp);
  }
```

Observație: *Unele compilatoare de C dispun, de asemenea, de funcția* ***getpsp****, care returnează adresa de segment PSP:*

```
#include <dos.h>
unsigned getpsp(void);
```

732 UTILIZAREA MODIFICATORULUI CONST ÎN DECLARAREA VARIABILELOR

C/C++

Câteva dintre prototipurile de funcții pe care le prezintă această carte utilizează cuvântul cheie *const* înainte de numele parametrilor:

```
char *strcpy(char *destinatie, const char *sursa);
```

Atunci când utilizați cuvântul cheie *const* înaintea numelor parametrilor, indicați compilatorului că programul nu trebuie să modifice parametrul în cadrul funcției. Dacă o instrucțiune încearcă să modifice parametrul, compilatorul va genera un mesaj de eroare. C permite, de asemenea, utilizarea cuvântului cheie *const* atunci când declarați variabile. Când declarați o variabilă ca o constantă, compilatorul de C va executa inițializarea variabilei în același moment. După inițializare, compilatorul va genera o eroare de fiecare dată când încercați să modificați constanta. Următoarea instrucțiune creează câteva constante diferite:

```
const int numar = 1001;
const float pret = 39.95;
```

Avantajul utilizării constantelor în locul unei macrocomenzi pe care o creați cu *#define* este acela că puteți, utilizând constanta, să precizați în mod explicit tipul valorii.

Observație: *Atunci când declarați o constantă, puteți utiliza și un pointer alias pentru a modifica valoarea variabilei constante.*

733 UTILIZAREA TIPURILOR ENUMERATE

C/C++

După cum ați învățat, utilizând nume semnificative de variabile puteți crește semnificativ lizibilitatea programelor dumneavoastră. În plus, înlocuind valorile constante (cum ar fi 1, 2 și 3) cu nume semnificative care corespund valorilor pe care le reprezintă variabilele (cum ar fi luni, marți și miercuri), puteți mări lizibilitatea programelor dumneavoastră. Pentru a ajuta programele dumneavoastră să lucreze cu astfel de constante, C acceptă *tipurile enumerate*. În general, un tip enumerat este o listă de elemente care are fiecare valoare a sa unică. Puteți utiliza tipurile enumerate pentru a mări lizibilitatea programelor dumneavoastră. De exemplu, următoarea declarare creează un tip de *enumerare* numit *zile_saptam*:

```
enum zile_saptam { luni, marti, miercuri, joi, vineri };
```

Enumerarea este similară cu o definire de structură, în care puteți să declarați imediat variabile de tipul respectiv sau puteți să faceți referință la numele tipului mai târziu:

```
enum zile_saptam { luni, marti, miercuri, joi, vineri } zi_lucru;
enum zile_saptam zi_libera;
```

După ce declarați o variabilă de tip enumerare, puteți să faceți referință la un nume de membru pentru a atribui o valoare variabilei:

```
zi_libera = vineri;
zi_lucru = marti;
```

MODUL DE UTILIZARE A TIPULUI ENUMERARE

C/C++ 734

În secțiunea 733 ați învățat că programele dumneavoastră pot utiliza tipurile enumerare pentru a le mări lizibilitatea. Următorul program, *zile.c*, ilustrează modul în care programele dumneavoastră pot utiliza tipul enumerare pentru a le mări lizibilitatea:

```
#include <stdio.h>

void main(void)
  {
    enum { luni, marti, miercuri, joi, vineri } zi;
    for (zi = luni; zi <= vineri; zi++)
     if (zi == luni)
      printf("Fara haz--- intalniri toata ziua de luni\n");
     else if (zi == marti)
      printf("Fara haz---fac astazi treaba de luni\n");
     else if (zi == miercuri)
      printf("Ce zi...");
     else if (zi == joi)
      printf("Programul intalnirilor pentru urmatoarea zi de
            luni\n");
     else
      printf("Intalnire cu toti la petrecere!\n");
  }
```

O VALOARE ENUMERARE

C/C++ 735

Așa cum ați învățat în secțiunea 733, fiecare membru din cadrul unui tip de enumerare are o valoare unică. În mod implicit, compilatorul de C atribuie primului membru valoarea 0, celui de al doilea valoarea 1 și așa mai departe. Următorul program, *enumer.c*, afișează valorile care corespund zilelor enumerate ale săptămânii:

```
#include <stdio.h>

void main(void)
  {
    enum zile_saptam { luni, marti, miercuri, joi, vineri };
    printf("%d %d %d %d %d\n", luni, marti, miercuri, joi, vineri);
  }
```

Atunci când compilați și executați programul *enumer.c*, ecranul dumneavoastră va afișa valorile de la 0 la 4.

736 ATRIBUIREA UNEI ANUMITE VALORI PENTRU UN TIP ENUMERARE

C/C++

În secțiunea 733 ați învățat că C atribuie o valoare unică pentru fiecare membru al unui tip enumerare. În conformitate cu funcțiile executate de programul dumneavoastră, puteți să specificați valoarea fiecărui membru. Următoarea declarare, de exemplu, atribuie valorile 10, 20, 30, 40 și 50 zilelor săptămânii:

```
enum zile_saptam { luni = 10, marti = 20, miercuri = 30,
                   joi = 40, vineri = 50 };
```

Următorul program, *setenum.c*, atribuie aceste valori membrilor tipului enumerare și apoi afișează valoarea fiecărui membru:

```
#include <stdio.h>

void main(void)
  {
    enum zile_saptam { luni = 10, marti = 20, miercuri = 30,
      joi = 40, vineri = 50 };
    printf("%d %d %d %d %d\n", luni, marti, miercuri, joi, vineri);
  }
```

În plus față de atribuirea unei valori fiecărui membru, puteți să atribuiți, de asemenea, o valoare unui anumit membru. Compilatorul de C va incrementa valoarea fiecăruia dintre ceilalți membri cu 1. Următoarea declarare, de exemplu, atribuie valorile 10, 11, 12, 13 și 14 zilelor săptămânii:

```
enum zile_saptam { luni = 10, marti, miercuri, joi, vineri };
```

737 SALVAREA ȘI REFACEREA REGISTRELOR

C/C++

Multe compilatoare vă permit accesul la valorile de registre din cadrul programelor dumneavoastră în C. Programele care execută astfel de operații depun adesea valorile de registre în stivă înainte ca programul să modifice registrul și apoi extrag valoarea pentru a o reface în registru. Dacă aveți o funcție care execută astfel de operații de nivel inferior, puteți să indicați compilatorului de C să insereze instrucțiunile PUSH și POP în codul obiect. PUSH și POP vor salva automat toate registrele atunci când programul apelează funcția și va restabili apoi registrele, înainte ca funcția să-și încheie execuția. Pentru a cere compilatorului să execute aceste operații, includeți, pur și simplu, modificatorul *_saveregs* în antetul funcției:

```
int _saveregs o_functie(int parametru);
```

Pentru a înțelege mai bine procesarea pe care modificatorul *_saveregs* o cere compilatorului, creați o funcție simplă care utilizează modificatorul *_saveregs* și apoi generați o listare a codului sursă în limbaj de asamblare.

PREZENTAREA LISTELOR DINAMICE

C/C++ 738

În capitolul despre structuri al acestei cărţi aţi învăţat cum se grupează informaţiile corelate într-o singură variabilă. Dacă programul dumneavoastră trebuie să lucreze cu un număr fix de apariţii de structuri, programul poate crea o matrice de structuri. Cum însă programele dumneavoastră devin mai complexe, uneori poate că nu veţi cunoaşte dinainte de câte intrări de structuri aveţi nevoie. În aceste cazuri, aveţi două opţiuni. În prima, programul dumneavoastră poate aloca memorie dinamică pentru matricea de structuri. În a doua, programul dumneavoastră poate crea o *listă înlănţuită* de structuri, unde o intrare indică următoarea intrare. Figura 738 ilustrează o listă înlănţuită de nume de fişiere.

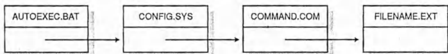


Figura 738 *O listă înlănţuită de nume de fişiere.*

În general, programul menţine un pointer la începutul listei. Un pointer la *NULL* indică ultima intrare în listă.

DECLARAREA UNEI STRUCTURI LISTĂ ÎNLĂNŢUITĂ

C/C++ 739

Pentru a crea o listă înlănţuită, unul dintre membrii structurii trebuie să fie un pointer la o structură de acelaşi tip. De exemplu, să considerăm următoarea structură:

```
struct Listfis
  {
    char numefis[64];
    struct Listfis *urmator;
  };
```

Membrul *numefis* conţine un nume de fişier. Membrul *urmator* este un pointer la următoarea intrare în listă. Pentru a crea şi apoi pentru a trece printr-o listă înlănţuită, programul dumneavoastră va utiliza de obicei cel puţin două variabile. Variabila *start* este o structură. Membrul său *urmator* conţine un pointer la începutul listei sau *NULL* dacă lista este vidă. Variabila *nod* este un pointer la modul curent:

```
struct Listfis start, *nod;
```

CONSTRUIREA UNEI LISTE ÎNLĂNŢUITE

C/C++ 740

Pentru a crea o listă înlănţuită, programul dumneavoastră trebuie să execute următorii paşi:

1. Să declare structura care defineşte intrările listei.
2. Să declare variabilele *start* şi **nod*.
3. Să atribuie variabilei *start.urmator* valoarea *NULL* pentru a indica o listă vidă.

Pentru fiecare intrare în listă, programul dumneavoastră trebuie să execute următorii pași:

1. Să găsească sfârșitul listei astfel ca *nod->urmator* să fie *NULL*.
2. Să aloce memorie pentru noua intrare și să atribuie valoarea locației de start a memoriei la membrul pointer *nod->urmator*.
3. Să atribuie lui *nod* valoarea lui *nod->urmator*.
4. Să atribuie valorile membrilor la *nod*.
5. Să atribuie membrului *nod->urmator* valoarea *NULL* pentru indicarea noului sfârșit de listă.

741 UN EXEMPLU SIMPLU DE LISTĂ ÎNLĂNȚUITĂ



În secțiunea 740 ați învățat pașii pe care programele dumneavoastră trebuie să-i efectueze pentru a crea o listă înlănțuită. Următorul program, *1-10list.c*, creează o listă înlănțuită ale cărei intrări conțin numerele de la 1 la 10:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    int i;
    struct IntrariLista
      {
        int numar;
        struct IntrariLista *urmator;
      } start, *nod;
    start.urmator = NULL; // Lista vida
    nod = &start; // Indica inceputul listei
    for (i = 1; i <= 10; i++)
      {
        nod->urmator = (struct IntrariLista *)
          malloc(sizeof(struct IntrariLista));
        nod = nod->urmator;
        nod->numar = i;
        nod->urmator = NULL;
      }
    // Afiseaza lista
    nod = start.urmator;
    while (nod)
      {
        printf("%d ", nod->numar);
        nod = nod->urmator;
      }
  }
```

TRECEREA PRINTR-O LISTĂ ÎNLĂNȚUITĂ

C/C++ 742

În secțiunea 741, ați scris programul *1-10list.c*, care creează o listă înlănțuită simplă cu intrări care conțin numerele de la 1 la 10. Programul utilizează următoarea buclă pentru a afișa intrările listei:

```
    // Afiseaza lista
    nod = start.urmator;
    while (nod)
    {
      printf("%d ", nod->numar);
      nod = nod->urmator;
    }
```

În cadrul acestui fragment de cod, variabila *start.urmator* indică prima intrare din listă. După cum vedeți, codul atribuie adresa primei intrări la *nod*. De asemenea, așa cum vă veți aminti, *NULL* indică sfârșitul listei. De aceea, bucla testează doar valoarea curentă a lui *nod* pentru a vedea dacă este *NULL*. Dacă *nod* nu este *NULL*, bucla va afișa valoarea intrării și va atribui lui *nod* adresa următoarei intrări în listă.

CONSTRUIREA UNEI LISTE MAI UTILE

C/C++ 743

În secțiunea 741 ați creat o listă înlănțuită simplă conținând numerele de la 1 la 10. Următorul program, *list_ fis.c*, creează o listă înlănțuită care conține toate numele de fișiere din directorul curent:

```
#include <stdio.h>
#include <dirent.h>
#include <alloc.h>
#include <string.h>

void main(int argc, char *argv[])
  {
    DIR *pointer_director;
    struct dirent *intrare;
    struct ListFis
      {
        char numefis[64];
        struct ListFis *urmator;
      } start, *nod;

    if ((pointer_director = opendir(argv[1])) == NULL)
      printf("Eroare la deschiderea %s\n", argv[1]);
    else
      {
        start.urmator = NULL;
        nod = &start;
        while (intrare = readdir(pointer_director))
          {
```

```
            nod->urmator = (struct ListFis *)
            malloc(sizeof(struct ListFis));
            nod = nod->urmator;
            strcpy(nod->numefis, intrare);
            nod->urmator = NULL;
          }
        closedir(pointer_director);
        nod = start.urmator;
        while (nod)
          {
            printf("%s\n", nod->numefis);
            nod = nod->urmator;
          }
      }
  }
```

Programul *list_fis.c* utilizează funcția *readdir* pentru a citi intrările directorului. Programul alocă apoi memorie pentru a păstra intrarea și copiază numele de fișier corespunzător intrării în listă. După ce programul adaugă toate fișierele la listă, el va cicla prin listă și va afișa fiecare intrare.

744 ADĂUGAREA UNEI INTRĂRI ÎN LISTĂ

C/C++

Fiecare program de listă înlănțuită pe care l-a prezentat această carte până acum, a construit o listă înlănțuită întreagă dintr-o dată, de obicei în cadrul unei bucle *while* sau *for*. În funcție de programul dumneavoastră, probabil că la un moment dat veți dori să adăugați intrări la listă. Cea mai simplă cale de adăugare a unei intrări este adăugarea intrării la sfârșitul listei. Pentru a adăuga un element la sfârșitul unei liste înlănțuite, ciclați prin listă până veți găsi elementul care conține membrul *urmator*, care indică *NULL*:

```
nod = &start;
while (nod->urmator)
    nod = nod->urmator;
```

Atunci când *nod ->urmator* indică *NULL*, ați găsit sfârșitul listei și puteți deci să alocați memorie pentru noua intrare:

```
nod->urmator = malloc(dim_necesara);
```

Apoi, atribuiți intrării valoarea pe care o doriți (în cadrul elementului *membru*) și atribuiți câmpul *urmator* al noii intrări să indice *NULL*:

```
nod = nod->urmator;
nod->membru = o_valoare;
nod->urmator =NULL;
```

În anumite cazuri, programul dumneavoastră poate să plaseze elementele la o anumită locație într-o listă. Veți învăța cum se plasează elementele în secțiunea 745.

INSERAREA UNEI INTRĂRI ÎN LISTĂ

C/C++745

În secţiunea 744 aţi învăţat cum se adaugă un element la sfârşitul unei liste înlănţuite. În conformitate cu funcţia programului dumneavoastră, puteţi să plasaţi elementele la anumite locaţii într-o listă. De exemplu, dacă vreţi să creaţi o listă înlănţuită care conţine numele sortate ale fişierelor din directorul curent, atunci programul dumneavoastră trebuie să plaseze fiecare nume de fişier într-o listă, la poziţia corectă. Pentru a insera un element la o anumită locaţie într-o listă, programul dumneavoastră va urmări de obicei valoarea *nod* de început, pe cea curentă şi pe cea precedentă. Atunci când programul dumneavoastră trebuie să insereze un nou element, el va executa următoarea procedură:

```
struct MembruLista start, *nod, *precedent, *nou;
// Cod care executa inserarea unei intrari
// intre elementele indicate de nod si precedentul
nou = malloc(sizeof(struct MembruLista));
nou->urmator = nod;
precedent->urmator = nou;
nou->membru = o_valoare;
```

AFIŞAREA UNUI DIRECTOR SORTAT

C/C++746

În secţiunea 745 aţi învăţat că pentru a insera un element într-o listă simplu înlănţuită (unde fiecare element conţine un pointer la următorul element), programele dumneavoastră trebuie să urmărească nodul curent şi cel precedent (elementele listei). Următorul program, *sortlist.c*, inserează elemente într-o listă pentru a crea o listă care conţine numele sortate ale fişierelor din directorul curent:

```
#include <stdio.h>
#include <dirent.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    DIR *pointer_director;
    struct dirent *intrare;
    struct ListFis
      {
        char numefis[64];
        struct ListFis *urmator;
      } start, *nod, *precedent, *nou;
    if ((pointer_director = opendir(argv[1])) == NULL)
      printf("Eroare la deschiderea %s\n", argv[1]);
    else
      {
        start.urmator = NULL;
        while (intrare = readdir(pointer_director))
```

```
        {
          // Gaseste locatia corecta
          precedent = &start;
          nod = start.urmator;
          while ((nod) && (strcmp(intrare, nod->numefis) > 0))
            {
              nod = nod->urmator;
              precedent = precedent->urmator;
            }
          nou = (struct ListFis *)
          malloc(sizeof(struct ListFis));
          if (nou == NULL)
            {
              printf("Insuficienta memorie pentru a stoca
                        lista\n");
              exit(1);
            }
          nou->urmator = nod;
          precedent->urmator = nou;
          strcpy(nou->numefis, intrare);
        }
      closedir(pointer_director);
      nod = start.urmator;
      while (nod)
        {
          printf("%s\n", nod->numefis);
          nod = nod->urmator;
        }
    }
}
```

747 ȘTERGEREA UNUI ELEMENT DINTR-O LISTĂ C/C++

În secțiunea 745 ați învățat cum se inserează un element într-o listă simplu înlănțuită. Pentru că programele dumneavoastră operează cu liste înlănțuite, probabil că va trebui uneori să ștergeți un element dintr-o listă. Eliminarea unui element dintr-o listă separată este foarte asemănătoare cu operația de inserare, în care trebuie să urmăriți pointerii la nodurile curent și precedent. După ce programul localizează elementul din listă pe care doriți să îl ștergeți, puteți utiliza un cod similar cu următorul pentru a elimina nodul:

```
precedent->urmator = nod->urmator;
free(nod);
```

Următorul program, *elimin5.c*, creează o listă înlănțuită care conține numerele de la 1 la 10. Programul caută apoi în listă elementul care conține numărul cinci. Apoi, programul elimină acel element:

```
#include <stdio.h>
#include <alloc.h>
void main(void)
  {
    int i;
    struct IntrareLista
      {
        int numar;
        struct IntrareLista *urmator;
      } start, *nod, *precedent;
    start.urmator = NULL; // Lista vida
    nod = &start; // Indica startul listei
    for (i = 1; i <= 10; i++)
      {
        nod->urmator = (struct IntrareLista *)
          malloc(sizeof(struct IntrareLista));
        nod = nod->urmator;
        nod->numar = i;
        nod->urmator = NULL;
      }
    nod = start.urmator; // Elimina numarul 5
    precedent = &start;
    while (nod)
      if (nod->numar == 5)
        {
          precedent->urmator = nod->urmator;
          free(nod);
          break; // Incheie bucla
        }
      else
        {
          nod = nod->urmator;
          precedent = precedent->urmator;
        }
    nod = start.urmator; // Afiseaza lista
    while (nod)
      {
        printf("%d ", nod->numar);
        nod = nod->urmator;
      }
  }
```

UTILIZAREA LISTEI DUBLU ÎNLĂNȚUITE

C/C++ 748

O *listă simplu înlănțuită* este astfel numită pentru că fiecare element al listei conține un pointer la următorul element. Ați învățat că pentru a insera un element într-o listă simplu

înlănțuită, programele dumneavoastră trebuie să mențină pointeri la elementele curent și precedent. Pentru a simplifica procesul de inserare și de eliminare a elementelor dintr-o listă, programele dumneavoastră pot utiliza o *listă dublu înlănțuită.* Într-o listă dublu înlănțuită, fiecare element menține un pointer la următorul și la precedentul element. Figura 748 ilustrează o listă dublu înlănțuită.

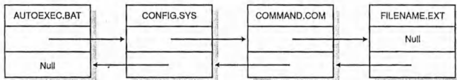


Figura 748 *O listă dublu înlănțuită menține doi pointeri.*

Următoarea structură ilustrează structura unei liste dublu înlănțuite:

```
struct ListFis
   {
     char numefis[64];
     struct ListFis *urmator;
     struct ListFis *precedent;
   };
```

Atunci când programele dumneavoastră utilizează lista dublu înlănțuită, programul poate traversa prin elementele listei de la stânga la dreapta sau de la dreapta la stânga. De aceea, lista trebuie să mențină doi pointeri *NULL.* Atunci când programul traversează lista de la stânga la dreapta, trebuie să sesizeze că a ajuns la sfârșitul listei când *nod->urmator* este *NULL.* De asemenea, atunci când programul traversează lista de la dreapta la stânga, trebuie să sesizeze că a ajuns la sfârșitul listei când *nod->precedent* este *NULL,* ceea ce indică sfârșitul listei.

749 CONSTRUIREA UNEI LISTE DUBLU ÎNLĂNȚUITE SIMPLE

C/C++

În secțiunea 748 ați învățat că o listă dublu înlănțuită simplifică procesul de inserare și de eliminare a elementelor listei. Următorul program, *dbl_1_10.c,* utilizează o listă dublu înlănțuită pentru a afișa numerele de la 1 la 10 și apoi invers:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    int i;
    struct IntrareLista
      {
        int numar;
        struct IntrareLista *urmator;
```

```
        struct IntrareLista *precedent;
      } start, *nod;
    start.urmator = NULL; // Lista vida
    start.precedent = NULL;
    nod = &start; // Indica inceputul listei
    for (i = 1; i <= 10; i++)
      {
        nod->urmator = (struct IntrareLista *)
        malloc(sizeof(struct IntrareLista));
        nod->urmator->precedent = nod;
        nod = nod->urmator;
        nod->numar = i;
        nod->urmator = NULL;
      }
    nod = start.urmator; // Afiseaza lista
    do
      {
        printf("%d ", nod->numar);
        nod = nod->urmator;
      } while (nod->urmator); // Arata 10 numai o data
    do
      {
        printf("%d ", nod->numar);
        nod = nod->precedent;
      } while (nod->precedent);
  }
```

NOD->PRECEDENT->URMATOR

C/C++ 750

După cum aţi învăţat, lucrul cu listele dublu înlănţuite simplifică operaţiile de inserare şi eliminare a elementelor listei. Urmărind cu atenţie programele care lucrează cu liste dublu înlănţuite, veţi putea întâlni instrucţiuni cum ar fi următoarea:

```
nod->precedent->urmator = nod_nou;
```

Dacă studiaţi această instrucţiune, începeţi de la stânga la dreapta. Figura 750 ilustrează modul în care compilatorul de C rezolvă pointerii.

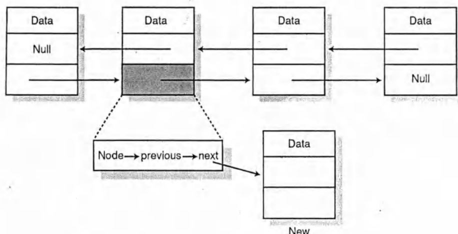


Figura 750 *Rezolvarea unei operații complexe cu pointeri.*

751 ELIMINAREA UNUI ELEMENT DINTR-O LISTĂ DUBLU ÎNLĂNȚUITĂ

C/C++

O listă dublu înlănțuită simplifică procesul de inserare și eliminare a elementelor listei. Următorul program, *elimin_7.c*, construiește o listă dublu înlănțuită care conține numerele de la 1 la 10. Programul caută apoi în listă intrarea care conține numărul 7 și elimină intrarea:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    int i, gasit;
    struct IntrareLista {
      int numar;
      struct IntrareLista *urmator;
      struct IntrareLista *precedent;
    } start, *nod;
    start.urmator = NULL; // Lista vida
    start.precedent = NULL;
    nod = &start; // Indica inceputul listei
    for (i = 1; i m,<= 10; i++)
      {
        nod->urmator = (struct IntrareLista *)
          malloc(sizeof(struct IntrareLista));
        nod->urmator->precedent = nod;
        nod = nod->urmator;
        nod->numar = i;
```

```
            nod->urmator = NULL;
        }
    // elimina intrarea
    nod = start.urmator;
    gasit = 0;
    do {
        if (nod->numar == 7)
          {
            gasit = 1;
            nod->precedent->urmator = nod->urmator;
            nod->urmator->precedent = nod->precedent;
            free(nod);
          }
        else
          nod = nod->urmator;
    } while ((nod) && (! gasit)); // Arata 10 numai o data
    nod = start.urmator;
    do {
        printf("%d ", nod->numar);
        nod = nod->urmator;
    } while (nod);
  }
```

INSERAREA UNUI ELEMENT ÎNTR-O LISTĂ DUBLU ÎNLĂNŢUITĂ

C/C++ 752

Aşa cum aţi învăţat, o listă dublu înlănţuită simplifică inserarea şi eliminarea elementelor listei. Următorul program, *lst_1_10.c*, construieşte o listă care conţine numerele 1, 3, 5, 7 şi 9. Programul inserează apoi numerele 2, 4, 6, 8 şi 10 la locaţia corectă în cadrul listei:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
    int i;
    struct IntrareLista {
      int numar;
      struct IntrareLista *urmator;
      struct IntrareLista *precedent;
    } start, *nod, *nou;
    start.urmator = NULL; // Lista vida
    start.precedent = NULL;
    nod = &start; // Indica inceputul listei
    for (i = 1; i < 10; i += 2)
      {
        nod->urmator = (struct IntrareLista *)
```

```
      malloc(sizeof(struct IntrareLista));
    nod->urmator->precedent = nod;
    nod = nod->urmator;
    nod->numar = i;
    nod->urmator = NULL;
  }
  for (i = 2; i <= 10; i += 2)
  {
    int gasit = 0;
    nou = (struct IntrareLista *)
      malloc(sizeof(struct IntrareLista));
    nou->numar = i;
    nod = start.urmator;
    do {
      if (nod->numar > nou->numar)
       {
         nou->urmator = nod;
         nou->precedent = nod->precedent;
         nod->precedent->urmator = nou;
         nod->precedent = nou;
         gasit = 1;
       }
      else
       nod = nod->urmator;
      } while ((nod->urmator) && (! gasit));
      if (! gasit)
       if (nod->numar > nou->numar)
        {
          nou->urmator = nod;
          nou->precedent = nod->precedent;
          nod->precedent->urmator = nou;
          nod->precedent = nou;
        }
      else
        {
          nou->urmator = NULL;
          nou->precedent = nod;
          nod->urmator = nou;
        }
  }
  // Afiseaza lista
  nod = start.urmator;
  do {
    printf("%d ", nod->numar);
    nod = nod->urmator;
  } while (nod);
}
```

PROCESELE COPIL

C/C++ 753

Atunci când programele dumneavoastră rulează, un program poate să ruleze un al doilea program, numit *proces copil*. Programul care rulează cel de al doilea program este denumit program *părinte*. În funcție de necesitățile dumneavoastră, procesul copil poate să ruleze până la încheiere și apoi programul părinte poate continua sau procesul copil poate lua locul procesului părinte, suprascriind programul părinte în memorie. Atunci când programul copil rulează până la încheiere, apoi programul părinte continuă, execuția programului copil se numește *spawning*. Atunci când procesul copil înlocuiește procesul părinte în memorie, programul trebuie să apeleze funcții de bibliotecă run-time de tip *exec* pentru procesul copil. Pentru a facilita executarea acestor procese în cadrul programelor dumneavoastră, biblioteca run-time de C dispune de două tipuri de funcții: *spawn* și *exec*. Secțiunile 754 și 757 abordează aceste rutine de bibliotecă run-time în detaliu.

UTILIZAREA FUNCȚIILOR DE TIP SPAWN PENTRU UN PROCES COPIL

C/C++ 754

Așa cum ați învățat în secțiunea 753, atunci când un program apelează o funcție *spawn* pentru un task copil, programul părinte își suspendă execuția cât timp rulează procesul copil, pentru ca apoi să continue. Pentru a apela o funcție *spawn* pentru un proces copil, programele dumneavoastră pot utiliza funcția *spawnl*:

```
#include <process.h>
#include <stdio.h>

int spawnl(int mod, char *copil, char *arg0, ... , char *argn,
  NULL);
```

Parametrul *copil* este un pointer la un șir de caractere care specifică numele fișierului executabil care conține procesul copil. Parametrii *arg0* până la *argn* specifică argumentele liniei de comandă a procesului copil. Parametrul *mod* specifică modul în care programul dumneavoastră rulează procesul copil. Tabelul 754.1 listează valorile posibile pentru mod.

Valoare	Mod de execuție
P_NOWAIT	Procesul părinte continuă execuția în paralel cu procesul copil (nu este posibil pentru programele sub DOS)
P_OVERLAY	Procesul copil suprascrie procesul părinte în memorie
P_WAIT	Procesul părinte se reia după ce procesul copil se încheie

Tabelul 754.1 *Modurile de execuție ale procesului copil.*

Dacă funcția *spawnl* reușește, ea va returna valoarea 0. Dacă apare o eroare, funcția va returna valoarea -1 și va atribui variabilei globale *errno* una dintre valorile listate în tabelul 754.2.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 754.2 *Valorile de eroare returnate de funcția* ***spawnl***.

Pentru a înțelege mai bine procesele copil, creați programul *copil.c*, care va afișa argumentele liniei sale de comandă și intrările de mediu:

```
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[], char *mediu[])
  {
    printf("Linie de comanda\n");
    while (*argv)
      puts(*argv++);
    printf("Intrari de mediu\n");
    while (*mediu)
      puts(*mediu++);
  }
```

Compilați programul. Apoi, creați programul *spawnl.c*, care utilizează funcția *spawnl* pentru a executa procesul copil:

```
#include <process.h>
#include <stdio.h>

void main(void)
  {
    printf("Pe punctul de a apela procesul copil\n\n");
    spawnl(P_WAIT, "COPIL.EXE", "COPIL.EXE", "AAA", "BBB", "CCC",
         NULL);
    printf("\n\nRevine din procesul copil\n");
  }
```

Atunci când executați programul *spawnl.c*, ecranul dumneavoastră va afișa un mesaj care afirmă că este pe punctul de a apela procesul copil. Apoi, procesul copil va rula, afișând argumentele liniei sale de comandă și intrările de mediu. După ce procesul copil se încheie, programul va afișa un mesaj care afirmă că a revenit din procesul copil.

755 UTILIZAREA ALTOR FUNCȚII SPAWNLXX



În secțiunea 754 ați învățat că funcția *spawnl* vă permite rularea procesului copil. Dacă analizați biblioteca run-time de C, veți găsi câteva alte funcții *spawnlxx*, cum arătăm mai jos:

```
#include <process.h>
#include <stdio.h>
int spawnle(int mod, char *copil, char *arg0, ... , char *argn,
            NULL, char *mediu);
int spawnlp(int mod, char *copil, char *arg0, ... , char *argn,
            NULL);
int spawnlpe(int mod, char *copil, char *arg0, ... , char *argn,
            NULL, char *mediu);
```

Dacă oricare dintre funcțiile *spawnlxx* reușește, funcția va returna valoarea 0. Dacă apare o eroare, fiecare dintre funcții va returna valoarea –1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 755.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 755 *Valorile de eroare returnate de funcțiile* ***spawnlxx***.

Parametrii funcțiilor *spawnlxx* sunt similari celor pe care îi utilizează funcția *spawnl*, cum se arată în secțiunea 754. Însă, funcțiile *spawnlxx* utilizează și parametrul *mediu*, care conține un pointer către intrările de mediu ale procesului copil. Diferența între funcțiile *spawnl* și *spawnlp* este aceea că *spanwlp* și *spawnlpe* vor căuta calea comenzii pentru procesul copil. Următorul program, *spawnlxx.c*, ilustrează utilizarea funcțiilor *spawnlxx*:

```
#include <process.h>
#include <stdio.h>

void main(void)
  {
    char *mediu[] = { "FILE=SPAWNLXX.C", "LANGUAGE=C", "OS=DOS",
                      NULL};
    spawnle(P_WAIT, "COPIL.EXE", "COPIL.EXE", "Utilizeaza-spawnle",
              "BBB", NULL, mediu);
    spawnlp(P_WAIT, "COPIL.EXE", "COPIL.EXE", "Utilizeaza-spawnlp",
              "BBB", NULL);
    spawnlpe(P_WAIT, "COPIL.EXE", "COPIL.EXE", "Utilizeaza-spawnlpe",
            "BBB", NULL, mediu);
  }
```

UTILIZAREA FUNCȚIILOR DE TIP SPAWNVXX



În secțiunea 754 ați învățat cum se utilizează funcția *spawnl* pentru a crea un proces copil. De asemenea, în secțiunea 755 ați utilizat diferite funcții *spawnlxx*, care vă permit

transmiterea unei matrice de intrări de mediu către procesul copil. Funcțiile *spawnlxx* vă permit, de asemenea, să utilizați calea comenzii pentru a localiza procesul copil. Atunci când utilizați funcțiile *spawnl*, transmiteți argumentele liniei de comandă ca o listă de parametri terminată în *NULL*. În plus față de funcțiile *spawnlxx*, C dispune de o serie de funcții *spawnvxx*, care vă permit transmiterea parametrilor liniei de comandă ca o matrice de șiruri de caractere:

```
#include <stdio.h>
#include <process.h>

int spawnv(int mod, char *copil, char *argv[]);
int spawnve(int mod, char *copil, char *argv[], char *mediu[]);
int spawnvp(int mod, char *copil, char *argvp[]);
int spawnvpe(int mod, char *copil, char *argv[], char *mediu[]);
```

Dacă oricare din funcțiile *spawnvxx* reușesc, ele vor returna valoarea 0. Dacă apare o eroare, funcțiile vor returna valoarea –1 și vor stabili variabila globală *errno* la una dintre valorile listate în tabelul 756.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 756 *Valorile de eroare returnate de funcțiile* ***spawnvxx***.

Parametrii funcțiilor *spawnvxx* sunt similari celor pe care îi utilizează funcția *spawnlxx*, cu excepția că funcțiile *spawnvxx* transmit argumentele liniei de comandă ca o matrice de șiruri de caractere. Următorul program, *spawnvxx.c*, ilustrează funcțiile *spawnvxx*:

```
#include <stdio.h>
#include <process.h>

void main(void)
  {
    char *mediu[] = { "FILENAME=SPAWNVXX.C", "OS=DOS",
                      "ROUTINES=SPAWNVXX", NULL };
    char *argv[] = { "COPIL.EXE", "AAA", "BBB", NULL };
    spawnv(P_WAIT, "COPIL.EXE", argv);
    spawnve(P_WAIT, "COPIL.EXE", argv, mediu);
    spawnvp(P_WAIT, "COPIL.EXE", argv);
    spawnvpe(P_WAIT, "COPIL.EXE", argv, mediu);
  }
```

UTILIZAREA FUNCȚIILOR DE TIP EXEC PENTRU UN PROCES COPIL

C/C++ 757

După cum ați învățat în secțiunea 753, când un program apelează o funcție de tip *exec* pentru un task copil, procesul copil suprascrie programul părinte în memorie. Deoarece procesul copil suprascrie programul părinte, procesul părinte nu se mai reia niciodată. Pentru apelarea funcțiilor *exec*, programele dumneavoastră pot utiliza funcția *execl*:

```
#include <process.h>
#include <stdio.h>

int execl(char *copil, char *arg0, ... , char *argn, NULL);
```

Parametrul *copil* este un pointer la șirul de caractere care specifică numele fișierului executabil conținând procesul copil. Parametrii de la *arg0* la *argn* specifică argumentele liniei de comandă a procesului copil.

Dacă funcția *execl* reușește, ea nu va returna o valoare. Dacă apare o eroare, funcția va returna valoarea -1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 757.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 757 *Valorile de eroare returnate de funcția* ***execl***.

Pentru a înțelege mai bine utilizarea funcțiilor *exec*, creați următorul program, *copil.c*, care afișează argumentele liniei de comandă și intrările de mediu:

```
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[], char *mediu[])
  {
   printf("Linia de comanda\n");
   while (*argv)
     puts(*argv++);
   printf("Intrarile de mediu\n");
   while (*mediu)
     puts(*mediu++);
  }
```

Compilați programul. Creați apoi următorul program, *execl.c*, care utilizează funcția *execl* pentru a executa procesul copil:

```
#include <process.h>
#include <stdio.h>

void main(void)
```

```
    {
      printf("Pe punctul de a apela procesul copil\n\n");
      execl("COPIL.EXE", "COPIL.EXE", "AAA", "BBB", "CCC", NULL);
      printf("\n\nRevine din procesul copil--AR TREBUI SA NU
        APARA\n");
    }
```

Atunci executați programul *execl.c*, ecranul va afișa un mesaj care anunță că programul urmează să apeleze procesul copil. După aceea, procesul copil va rula, afișând argumentele liniei de comandă și intrările de mediu. Procesul copil suprascrie procesul părinte, astfel că după încheierea procesului copil nu mai apare nici o altă procesare.

758 UTILIZAREA ALTOR FUNCȚII EXECLXX



În secțiunea 757 ați învățat că funcția *execl* vă permite rularea proceselor copil. Dacă examinați biblioteca run-time de C, veți găsi câteva alte funcții *execlxx*:

```
#include <stdio.h>
#include <process.h>

int execle(char *copil, char *arg0, ... , char *argn, NULL,
           char *mediu);
int execlp(char *copil, char *arg0, ... , char *argn, NULL);
int execlpe(char *copil, char *arg0, ... , char *argn, NULL,
            char *mediu);
```

Dacă oricare dintre funcțiile *execlxx* reușește, funcția respectivă nu va returna o valoare. Dacă apare o eroare, funcția va returna valoarea –1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 758.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 758 *Valorile de eroare returnate de funcțiile* ***execlxx***.

Parametrii acestor funcții sunt similari celor utilizați de funcția *spawnl*, cu excepția parametrului *mediu*, care conține un pointer la intrările de mediu ale procesului copil (secțiunea 754 descrie funcția *spawnl* în detaliu). Diferența dintre funcția *execl* și funcția *execlp* este aceea că funcțiile conținând litera p vor căuta calea comenzii pentru procesul copil. Următorul program, *execlpe.c*, ilustrează utilizarea funcției *execlpe*:

```
#include <process.h>
#include <stdio.h>

void main(void)
  {
```

```
    char *mediu[] = { "FILE=EXECLPE.C", "LANGUAGE=C", "OS=DOS",
                      NULL};
    execlpe("COPIL.EXE", "COPIL.EXE", "Utilizeaza-execlpe",
            "BBB", NULL, mediu);
  }
```

UTILIZAREA FUNCȚIILOR EXECVXX

C/C++759

În secțiunea 757 ați învățat cum să utilizați funcția *excl* pentru a crea un proces copil. De asemenea, în secțiunea 758 ați utilizat diferite funcții *execlxx*, care vă permit transmiterea unei matrice de intrări de mediu către procesul copil și vă permit, de asemenea, utilizarea căii comenzii pentru a localiza procesul copil. Atunci când utilizați funcții *execl*, transmiteți argumentele liniei de comandă ca o listă de parametri terminată în *NULL*. În plus față de funcțiile *execlxx*, compilatorul de C vă pune la dispoziție o serie de funcții *execvxx* care vă permite transmiterea parametrilor liniei de comandă ca o matrice de șiruri de caractere:

```
#include <stdio.h>
#include <process.h>

int execv(char *copil, char *argv[]);
int execve(char *copil, char *argv[], char *mediu[]);
int execvp(char *copil, char *argvp[]);
int execvpe(char *copil, char *argv[], char *mediu[]);
```

Dacă oricare dintre funcțiile *execvxx* reușește, funcția respectivă nu va returna o valoare. Dacă apare o eroare, funcția va returna valoarea –1 și va stabili variabila globală *errno* la una dintre valorile listate în tabelul 759.

Valoare	Descriere
E2BIG	Lista de argumente prea lungă
EINVAL	Argument nevalid
ENOENT	Nu găsește programul copil
ENOEXEC	Eroare de format
ENOMEM	Memorie insuficientă

Tabelul 759 *Valorile de eroare returnate de funcțiile* ***execvxx***.

Parametrii la funcțiile *execvxx* sunt similari celor pe care programul îi transmite funcțiilor *execlxx*, cu excepția că funcțiile *execvxx* transmit argumentele liniei de comandă ca o matrice de șiruri de caractere. Următorul program, *execvpe.c*, ilustrează funcția *execvpe*:

```
#include <stdio.h>
#include <process.h>

void main(void)
  {
    char *mediu[] = { "FILENAME=SPAWNVXX.C", "OS=DOS",
                      "ROUTINE=EXECVPE", NULL };
    char *argv[] = { "COPIL.EXE", "AAA", "BBB", NULL };
```

```
    execvpe("COPIL.EXE", argv, mediu);
}
```

760 EXTINDERILE DE PROGRAM

C/C++

După cum ați învățat în capitolul despre memorie al acestei cărți, sistemul DOS restricționează programele la primii 640Kb de memorie. În trecut, pentru a accepta programe mai mari, acestea își divizau codul în zone fixe, numite *extinderi* (*overlays*). În timp ce programul rula, el încărca diferite secțiuni de extindere, după cum era necesar. Deși extinderile permit programatorilor să scrie și să compileze programe foarte mari, ele cer programului să știe care dintre extinderi este încărcată la momentul curent, precum și care dintre extinderi conțin funcțiile dorite. Vă dați seama că un astfel de proces poate fi dificil, pentru că cere programelor de aplicații să dispună de operațiile de gestionare a memoriei pe care sistemele de operare le furnizează frecvent.

Pentru a ajuta programele dumneavoastră să încarce și să execute extinderile, sistemul DOS dispune de servicii de sistem care încarcă și execută fișiere și apoi transferă controlul la începutul fișierului. Puteți găsi dificil de utilizat sistemul DOS pentru gestionarea extinderilor. Multe compilatoare, însă, dispun de instrumente de administrare a extinderilor pe care programele dumneavoastră pot să le utilizeze pentru a gestiona extinderile. Pentru mai multe informații despre gestionarea memoriei, consultați documentația care însoțește compilatorul dumneavoastră. *Turbo C++ Lite* nu include propriul său gestionar de memorie.

761 ÎNTRERUPERILE

C/C++

O *întrerupere* este un eveniment care cauzează o oprire temporară a calculatorului din operația (*task*) executată curent, astfel că el poate să lucreze la o altă sarcină. Atunci când procesul de întrerupere se încheie, calculatorul reia operația inițială, ca și cum întreruperea nici nu ar fi avut loc. PC-ul acceptă întreruperile *hardware* și *software*. Capitolul despre DOS și BIOS al acestei cărți expune modul în care puteți utiliza întreruperile software pentru a accesa întreruperile DOS și BIOS. Pe de altă parte, dispozitive cum ar fi unitatea de disc sau ceasul de sistem al PC-ului generează întreruperi hardware. *Tratarea întreruperilor* este un software care răspunde la o anumită întrerupere. De obicei, programatorii experimentați scriu programe de tratare a întreruperilor în limbajul de asamblare. Însă, compilatoarele de C mai recente vă permit scrierea lor în C.

Primii 1.024 octeți ai memoriei PC-ului conțin adresele de segment și de deplasament (numite *vectori de întrerupere*) pentru 256 de întreruperi ale PC-ului. Atunci când apare o întrerupere anume, PC-ul depune în stivă pointerul de instrucțiune curent (IP), segmentul de cod (CS) și registrul *de indicatori* (starea mașinii). PC-ul găsește apoi adresa programului corespunzător de tratare a întreruperilor, utilizând vectorul de întrerupere. Tratarea întreruperii depune apoi registrele în stivă și începe prelucrarea. După ce tratarea întreruperii se încheie, el extrage registrele din stivă și apoi execută instrucțiunea IRET, care extrage registrul de indicatori, CS și IP. Următoarea instrucțiune în limbaj de asamblare, de exemplu, ilustrează o machetă tipică de tratare a întreruperii:

```
; salveaza registrele in stiva
PUSH AX
```

```
PUSH BX
PUSH CX
PUSH DX
PUSH SI
PUSH DI
PUSH DS
PUSH ES
; Executa instructiunile de tratare a intreruperii
; Extrage registrele din stiva
POP ES
POP DS
POP DI
POP SI
POP DX
POP CX
POP BX
POP AX
; Revine la operatia anterioara
IRET
```

Atunci când definiți propriul dumneavoastră program de tratare a întreruperii în limbaj de asamblare, actualizați apoi vectorul de întrerupere pentru a indica propria rutină de întrerupere. Înainte ca programul să se încheie, trebuie să restabiliți vectorul de întrerupere la valoarea sa inițială.

ÎNTRERUPERILE ÎN PC

C/C++762

Într-un PC, primii 1.024 octeți ai memoriei calculatorului dumneavoastră conțin adresele (vectori) a 256 de întreruperi PC. Sistemul de operare nu utilizează multe întreruperi, pe care le lasă la dispoziția programelor dumneavoastră pentru scopurile proprii. Tabelul 762 listează vectorii întreruperi PC și utilizarea lor.

Întrerupere	***Scop***	**Întrerupere**	***Scop***
00H	Împărțire hardware la 0	01H	Detectare hardware într-un pas
02H	Întrerupere nemascabilă	03H	Punct de întrerupere depanator
04H	Depășire aritmetică	05H	Tipărire ecran BIOS
08H	IRQ0 Bătaie de ceas	09H	IRQ1 Tastatură
0AH	IRQ2	0BH	IRQ3 COM2
OCH	IRQ4 COM2	0DH	IRQ5 PC/AT LPT1
OEH	IRQ6 Dischetă	0FH	IRQ7 LPT1
10H	Servicii video BIOS	11H	Listă echipamente BIOS
12H	Mărime memorie BIOS	13H	Servicii disc BIOS
14H	Servicii de comunicații BIOS	15H	Servicii diverse BIOS
16H	Servicii de tastatură BIOS	17H	Servicii imprimantă BIOS

(continuare)

Întrerupere	Scop	Întrerupere	Scop
18H	Invocare ROM-BASIC	19H	Reinițializare sistem
1AH	Ora BIOS	1BH	Program pentru CTRL-BREAK
1CH	Apelat de programul handler 08	1DH	Tabel parametri video
1EH	Tabel parametri disc	1FH	Tabel caractere grafice
20H	Program terminare DOS	21H	Servicii de sistem DOS
22H	Terminare program	23H	CTRL-BREAK DOS
24H	Eroare critică DOS	25H	Citire disc DOS
26H	Scriere disc DOS	27H	Terminare rezidentă DOS
28H	DOS inactiv	29H	*putchar* rapid DOS
2AH	Servicii MS-Net	2EH	Încărcător primar DOS
2FH	Multiplex MS-DOS	33H	Servicii *mouse*
40H	Vector dischetă	41H	Tabel parametri hard-disc
42H	Redirectare EGA BIOS	43H	Tabel parametri EGA
44H	Tabel de caractere EGA	4AH	Alarmă INT 70H PC/AT
5CH	Servicii NetBIOS	67H	Servicii EMS
70H	IRQ8 Timp real PC/AT	71H	IRQ9 Redirectare PC pentru INT 0AH
75H	IRQ13 Coprocesor matematic PC/AT		

Tabelul 762 *Vectorii de întrerupere ai PC-ului și utilizarea lor.*

763 *UTILIZAREA CUVÂNTULUI CHEIE INTERRUPT*



După cum ați învățat, sistemul DOS vă permite să creați propriile dumneavoastră programe de tratare a întreruperii. Dacă utilizați *Turbo C++ Lite*, cuvântul cheie *interrupt* face mai ușoară crearea tratării de întrerupere:

```
void interupt handler_propriu()
  {
     // Instructiunile din handler
  }
```

Atunci când compilatorul întâlnește cuvântul cheie *interrupt*, el inserează instrucțiunea de salvare și extragere a registrelor în stivă, după cum este nevoie, și apoi de revenire ulterioară din handler, utilizând instrucțiunea IRET (o instrucțiune în limbaj de asamblare). Pentru a înțelege procesul realizat de cuvântul cheie *interrupt*, creați un program care conține funcția *handler_propriu* sub forma limbajului de asamblare, dacă aveți un compilator care acceptă o astfel de compilare. Dacă examinați apoi fișierul sursă în limbaj de asamblare, veți observa instrucțiuni mașină similare celor prezentate în secțiunea 761.

DETERMINAREA UNUI VECTOR DE ÎNTRERUPERE

C/C++764

După cum aţi învăţat, un *vector de întrerupere* este o adresă de segment şi de deplasament a codului care tratează întreruperea. Pentru a ajuta programele dumneavoastră să determine un vector de întrerupere, multe compilatoare de mediu DOS dispun de funcţia *_dos_getvect*:

```
#include <dos.h>

void interrupt(*_dos_getvect(unsigned nr_intrp))();
```

Parametrul *nr_intrp* specifică numărul întreruperii dorite (de la 0 la 255). Funcţia returnează un pointer la tratarea întreruperii. Următorul program, *dos_vect.c*, va afişa vectorii pentru toate întreruperile PC-ului:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
    int i;
    for (i = 0; i <= 255; i++)
      printf("Intrerupere: %x Vector: %lx\n", i, _dos_getvect(i));
  }
```

STABILIREA UNUI VECTOR DE ÎNTRERUPERE

C/C++765

Atunci când programele dumneavoastră creează propriile lor programe de tratare a întreruperii, programele trebuie să atribuie vectorul întrerupere astfel încât să indice către propriul program de tratare a întreruperii. Pentru a ajuta programele dumneavoastră să atribuie vectorul de întrerupere, majoritatea compilatoarelor de mediu DOS dispun de funcţia *_dos_setvect*:

```
#include <dos.h>

void _dos_setvect(unsigned nr_intrp, void interrupt(*handler)());
```

Parametrul *nr_intrp* specifică întreruperea al cărui vector doriţi să îl modificaţi. Parametrul *handler* este un pointer la programul de tratare a întreruperii. Secţiunea 766 ilustrează utilizarea funcţiei *_dos_setvect*. Atunci când programele dumneavoastră modifică un vector de întrerupere, ele trebuie să salveze valoarea iniţială a vectorului, pentru a putea restabili vectorul iniţial înainte de încheierea programului. Dacă programul se încheie fără a restabili vectorul de întrerupere, sistemul dumneavoastră poate avea o conduită imprevizibilă (în general, poate să înceteze operarea).

766 ACTIVAREA ȘI DEZACTIVAREA ÎNTRERUPERILOR



Atunci când programele dumneavoastră execută instrucțiuni de întrerupere, probabil că veți dori uneori ca programele dumneavoastră să activeze și să dezactiveze întreruperile. Pentru a vă ajuta să controlați întreruperile, multe compilatoare de mediu DOS dispun de funcțiile macro *_disable* și *_enable*:

```
#include <dos.h>

void _disable(void);
void _enable(void);
```

Pentru a rula corect, PC-ul trebuie să genereze întreruperile de taste în mod regulat, astfel încât, dacă programul dumneavoastră dezactivează întreruperile, programele să minimizeze volumul de timp afectat pentru dezactivarea lor. De obicei, programele dumneavoastră vor utiliza funcțiile macro *_disable* și *_enable* atunci când modifică un vector de întrerupere cu *_dos_setvect*:

```
_disable();
_dos_setvect(nr_intrp, handler);
_enable();
```

767 CREAREA UNUI PROGRAM SIMPLU A ÎNTRERUPERII



După cum ați învățat, crearea programului de tratare a întreruperii cu cuvântul cheie *interrupt* din *Turbo C++ Lite* este mult mai facilă decât crearea lor în limbaj de asamblare. Următorul program, *nuptrscr.c*, creează un handler de întrerupere care îl înlocuiește pe cel BIOS *print-screen*, care tipărește conținutul ecranului atunci când apăsați combinația de taste SHIFT+PRTSC. Programul utilizează apoi funcția *_dos_getvect* pentru a determina valoarea vectorului inițial, pentru a-l putea restabili înainte de încheierea programului. Apăsarea combinației de taste SHIFT+PRTSC în timp ce programul dumneavoastră este activ, invocă propriul dumneavoastră program de tratare a întreruperii, care va afișa un mesaj pe ecran afirmând că ați apăsat SHIFT+PRTSC. Atunci când apăsați SHIFT+PRTSC de trei ori, programul *nuptrscr.c* se va încheia:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

int contor = 0;
void interrupt handler(void)
  {
    contor++;
  }
void main(void)
  {
    void interrupt (*handler_originar)();
```

```
    int vechi_contor = 0;
    handler_originar = _dos_getvect(5);
    _disable(); // Dezactiveaza intreruperile in timpul _dos_setvect
    _dos_setvect(5, handler);
    _enable();
    printf("Apasa SHIFT+PRTSC de trei ori sau orice tasta
      pentru a incheia\n");
    while (contor < 3)
      if (contor != vechi_contor)
        {
          printf("SHIFT+PRTSC apasate\n");
          vechi_contor = contor;
        }
      _disable();
      _dos_setvect(5, handler_originar);
      _enable();
  }
```

ÎNLĂNȚUIREA ÎNTRERUPERILOR

C/C++ 768

În secțiunea 767 ați învățat cum se scrie un program de tratare a întreruperii pentru operațiile de tipărire a ecranului din BIOS. În conformitate cu funcțiile executate de programul dumneavoastră, uneori poate că veți dori executarea tratării întreruperii originară după ce programul dumneavoastră își încheie prelucrarea. În astfel de situații, programul dumneavoastră poate să utilizeze funcția *_chain_interrupt*:

```
#include <dos.h>

void _chain_interrupt(void (interrupt far *handler)());
```

Următorul program, *contdos.c*, numără de câte ori apelează programul anumite întreruperi DOS (deci programul caută în registrul AH pentru INT 21). După încheierea programului *contdos.c*, acesta va afișa totalul numărului de servicii apelate de program:

```
#include <stdio.h>
#include <dos.h>
#include <dir.h>

int functie[255]; // Servicii DOS
void interrupt far (*handler_originar)();
void interrupt far handler(void)
  {
    char i;
    asm { mov i, ah }
    functie[i]++;
    _chain_intr(handler_originar);
  }
void main(void)
```

```
{
  int i;
  for (i = 0; i < 255; i++) // A duce la zero contorul functiilor
    functie[i] = 0;
  manipulator_originar = _dos_getvect(0x21);
  _disable();
  _dos_setvect(0x21, handler);
  _enable();
  printf("Acesta este un mesaj\n");
  fprintf(stdout, "Acesta este un al doilea mesaj\n");
  printf("Discul curent este %c\n", getdisk() + 'A');
  _disable();
  _dos_setvect(0x21, handler_originar);
  _enable();
  for (i = 0; i <= 255; i++)
    if (functie[i])
      printf("Functia %x apelata de %d ori\n", i, functie[i]);
}
```

769 GENERAREA UNEI ÎNTRERUPERI

C/C++

Așa cum ați învățat în capitolul despre DOS și BIOS al acestei cărți, biblioteca run-time de C dispune de funcțiile *intdos* și *int86*, care permit programelor dumneavoastră să acceseze serviciile DOS și BIOS. Atunci când programele dumneavoastră tratează anumite întreruperi, puteți să generați o întrerupere pentru a testa programul de tratare sau să invocați o anumită întrerupere. Pentru a genera o întrerupere, programele dumneavoastră pot utiliza funcția *geninterrupt*:

```
#include <dos.h>

void geninterrupt(int intrerupere);
```

Parametrul *intrerupere* specifică întreruperea pe care o doriți. Următorul program, *genintr.c*, apelează utilizarea ocazională a întreruperii 0xFF pentru a anunța programul de apariția unui anumit eveniment:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

void interrupt far (*handler_originar)();
void interrupt far handler(void)
  {
    printf("Tocmai a avut loc un eveniment\n");
    _disable();
    _dos_setvect(0xFF, handler_originar);
    _enable();
    exit(0);
  }
```

```
void main(void)
   {
     int i = 0;
     handler_originar = _dos_getvect(0xFF);
     _disable();
     _dos_setvect(0xFF, handler);
     _enable();
     while (i++ < 100)
       ;
     geninterrupt(0xFF);
   }
```

Observație: *Atunci când programați în Windows, veți urmări* ***mesajele*** *și* ***evenimentele****, mai degrabă decât să generați și să urmăriți întreruperile. Veți învăța mai multe despre mesaje și evenimente începând cu secțiunea 1251.*

DETECTAREA CRONOMETRULUI INTERN AL PC-ULUI

C/C++ 770

Multe dispozitive din interiorul sau exteriorul PC-ului trebuie să execute operații la anumite intervale de timp. Pentru a efectua aceste operații, PC-ul dispune de un cip cronometru care generează un semnal de 18,2 ori pe secundă. De fiecare dată când cronometrul semnalizează, PC-ul generează o întrerupere 8, care actualizează ceasul pentru ora curentă, și o întrerupere 1CH, pe care o pot detecta programele. Următorul program, *timer.c*, detectează întreruperea 1CH de fiecare dată când apare:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
int alfanum = 0;
int contor = 0;
void interrupt far handler(void)
   {
     if (++contor == 273) // 15 secunde
       {
         alfanum = !alfanum; // Comuta
         contor = 0;
       }
   }
void main(void)
   {
     int i;
     void interrupt far (*handler_originar)();
     handler_originar = _dos_getvect(0x1C);
     _disable();
     _dos_setvect(0x1c, handler);
     _enable();
```

```
    while (! kbhit())
      if (alfanum)
        for (i = 'A'; i <= 'Z'; i++)
          printf("%c\n", i);
      else
        for (i = 0; i <= 100; i++)
          printf("%d\n", i);
      _disable();
      _dos_setvect(0x1c, handler_originar);
      _enable();
  }
```

Programul de tratare a întreruperii totalizează numărul de apariții și comută valoarea variabilei globale *alfanum* la fiecare 15 secunde. Dacă valoarea variabilei *alfanum* este 1, programul va afișa în mod repetat literele alfabetului. Dacă valoarea lui *alfanum* este 0, programul va afișa numerele de la 1 la 100.

771 ERORILE CRITICE

C/C++

După cum probabil știți, atunci când încercați să utilizați o dischetă care nu conține un disc formatat, sistemul DOS va afișa un mesaj de eroare, urmat de obicei de:

```
Abort, Retry, Fail?
```

Astfel de erori sunt denumite *erori critice* pentru că sistemul DOS nu poate să le rezolve fără ajutorul utilizatorului. Atunci când apare o eroare critică, sistemul DOS invocă întreruperea 24H. Când programele dumneavoastră sesizează întreruperea 24H, ele pot să efectueze controlul erorilor critice și, posibil, chiar să afișeze un mesaj de eroare semnificativ sau cu instrucțiuni către utilizator. Atunci când DOS invocă INT 24H, el plasează un volum considerabil de informații în stivă, care descriu cauza și sursa erorii. După cum veți învăța în secțiunea 772, majoritatea compilatoarelor de C de mediu DOS dispun de funcții de bibliotecă run-time care facilitează controlul erorilor critice.

772 TRATAREA ERORILOR CRITICE ÎN C

C/C++

Așa cum descrie secțiunea 771, o eroare critică este o eroare de la care sistemul DOS nu poate continua fără intervenția utilizatorului. Pentru a ajuta programele dumneavoastră în C să trateze erorile critice, majoritatea compilatoarelor de C dispun de următoarele funcții de bibliotecă run-time:

```
#include <dos.h>

void _harderr(int (*handler)());
void _hardresume(int ax_registru);
void _hardreturn(int handler_val);
```

Funcția *_harderr* vă permite să specificați numele funcției care va trata erorile critice. Funcția *_hardresume* permite programelor dumneavoastră să returneze o valoare de stare către DOS. Funcția *_hardreturn*, pe de altă parte, permite returnarea unei valori (oricare ar fi ea), către program. Valoarea pe care funcția *_hardresume* o returnează trebuie să fie una dintre cele listate în tabelul 772.

Constantă	Semnificație
_HARDERR_ABORT	Încheiere a programului curent
_HARDERR_RETRY	Reîncercarea serviciului care a cauzat eroarea
_HARDERR_FAIL	Eșuarea serviciului care a cauzat eroarea
_HARDERR_IGNORE	Ignorarea erorii

Tabelul** 772 Constantele returnate de funcția **_hardresume.

Următorul program, *ersimpla.c*, pune la dispoziție un program simplu de tratare a erorii critice care va afișa un mesaj pe ecran și apoi va utiliza funcția *_hardresume* pentru a abandona programul:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void far handler(unsigned eroare_dispozitiv, unsigned cod_eroare,
                  unsigned far *antet_dispozitiv)
  {
    cputs("Eroarea critica incheie programul\n");
    _hardresume(_HARDERR_ABORT); // Abandoneaza
  }
void main(void)
  {
    FILE *pointer_fisier;
    _harderr(handler);
    pointer_fisier = fopen("A:UNFISIER.EXT", "r");
    printf("Mesaj program...\n");
    fclose(pointer_fisier);
  }
```

UN PROGRAM MAI COMPLET DE TRATARE A ERORII CRITICE

C/C++ 773

În secțiunea 772 ați creat un program simplu de tratare a erorilor critice, care afișează un mesaj și apoi încheie programul care a cauzat eroarea. Dacă studiați cu atenție programul, veți observa că acesta acceptă trei parametri:

```
void far handler(unsigned eroare_dispozitiv, unsigned cod_eroare,
                  unsigned far *antet_dispozitiv)
```

Atunci când sistemul DOS invocă un program de tratare a erorilor critice, el plasează informațiile despre eroare în stivă. Parametrul *eroare_dispozitiv* conține o valoare de eroare pe care DOS ar trebui, în mod obișnuit, să o transmită către programul de tratare în registrul AX. Dacă serviciul care a eșuat stabilește bitul 7 al parametrului *eroare_dispozitiv* la valoarea 1, eroarea este o eroare de disc. Tabelul 773.1 listează valorile pe care serviciul eșuat le poate atribui parametrului *eroare_dispozitiv*.

Bit(biți)	Valoare	Semnificație
0	0	Eroare citire
	1	Eroare scriere
1-2	00	Eroare DOS
	01	Eroare FAT
	10	Eroare director
	11	Eroare fișier
3	0	Eșuarea operației este nepermisă
	1	Eșuarea operației este permisă
4	0	Reîncercarea operației este nepermisă
	1	Reîncercarea operației este permisă
5	0	Ignorarea operației este nepermisă
	1	Ignorarea operației este permisă
7	0	Eroare de disc
	1	Nu este eroare de disc

Tabelul 773.1 *Valori de eroare pe care variabila* ***eroare_dispozitiv*** *le returnează.*

Parametrul *cod_eroare* conține informațiile de eroare pe care DOS le transmite, de obicei, programului de tratare a erorilor critice în registrul DI. Tabelul 773.2 listează valorile pe care DOS le transmite în registrul DI pentru erori de disc.

Valoare	Semnificație	Valoare	Semnificație
0	Protejat la scriere	1	Unitate necunoscută
2	Unitate nepregătită	3	Comandă necunoscută
4	Eroare de date CRC	5	Structură de cerere nevalidă
6	Eroare de poziționare	7	Tip de suport necunoscut
8	Sector negăsit	9	Imprimanta fără hârtie
10	Eroare de scriere	11	Eroare de citire
12	Eroare generală	15	Modificare de disc nevalidă

Tabelul 773.2 *Valorile de eroare de disc transmise de DOS în registrul DI.*

În sfârșit, parametrul *antet_dispozitiv* este un pointer la antetul driverului de dispozitiv pentru dispozitivul care a generat eroarea. Pentru a vă ajuta să înțelegeți mai bine cum pot utiliza programele dumneavoastră aceste valori, compact discul care însoțește această carte conține următorul program, *criterr.c*, care va afișa valorile conținute de aceste variabile.

```
int handler(int val_eroare,int ax,int bp,int si)
  {
    static char msg[80];
    unsigned di;
    int unitate, nr_eroare;
    di= _DI;  // daca nu este o eroare de disc atunci
              // un alt dispozitiv a avut probleme
```

```
   if (ax < 0)
     {
       error_win("Eroare de dispozitiv"); // raporteaza eroarea
       hardretn(ABORT);  // si se intoarce la program direct
                         // solicitand abandonarea
     }
   unitate= ax & 0x00FF; // altfel, a fost eroare de disc
   nr_eroare = di & 0x00FF; // raporteaza ce eroare a fost
   sprintf(msg, "Eroare: %s pe unitatea %c\r\nA)bort, R)etry,
     I)gnore: ", err_msg[nr_eroare], 'A' + unitate);
   hardresume(error_win(msg)); // se intoarce la program via
                               // intrerupere 0x23 DOS
       // cu abort, retry sau ignore introduse de utilizator.
   return ABORT;
 }
```

RESTABILIREA ÎNTRERUPERILOR DETERIORATE

C/C++ 774

Atunci când programele dumneavoastră se încheie, DOS restabilește automat valorile tratării de întrerupere CTRL+BREAK, tratării de terminare a programului și tratării erorilor critice la valorile lor dinaintea execuției programului dumneavoastră. În funcție de programele dumneavoastră, sistemul DOS poate să restabilească aceste valori înainte ca programul să se încheie. Pentru a restabili valorile, multe compilatoare dispun de funcția *_cexit*:

```
#include <process.h>
void _cexit(void);
```

Funcția *_cexit* nu încheie programul. În schimb, ea pur și simplu restabilește vectorii de întrerupere la care face referire paragraful anterior. Funcția nu va închide fișiere și nu va elibera buffere de disc. Când compilatorul dumneavoastră nu dispune de funcția *_cexit*, puteți să scrieți o funcție care restabilește întreruperile, utilizând valorile inițiale pe care programul dumneavoastră le-a salvat înainte de a modifica vectorii de întrerupere.

TRATAREA PENTRU CTRL+BREAK

C/C++ 775

În mod implicit, atunci când utilizatorul apasă combinația de taste CTRL+BREAK, programele dumneavoastră se încheie. Deseori, nu veți permite utilizatorului să poată încheia programul prin CTRL+BREAK. Ca o soluție, programele dumneavoastră pot defini un program de tratare a întreruperii utilizând funcția *ctrlbrk*:

```
#include <dos.h>
void ctrlbrk(int (*handler)(void));
```

Pentru a crea propriul dumneavoastră program de tratare a întreruperii pentru CTRL+BREAK, definiți o funcție pe care programul dumneavoastră poate să o apeleze de fiecare dată când utilizatorul apasă CTRL+BREAK și apoi transmiteți numele funcției către funcția *ctrlbrk*. Următorul program, *ctrlbrk.c*, creează un program propriu de tratare pentru CTRL+BREAK:

```
#include <stdio.h>
#include <dos.h>

int Handler_Ctrl(void)
  {
    printf("\007Apasa Enter pentru a incheia programul\n");
    return(1);
  }
void main(void)
  {
    Ctrlbrk(Handler_Ctrl);
    printf("Apasa Enter pentru a incheia programul\n");
    while (getchar() != '\n');
      ;
  }
```

Programul se repetă până când utilizatorul apasă ENTER. De fiecare dată când utilizatorul apasă CTRL+BREAK, programul invocă funcţia *Handlerr_Ctrl*. *Handler_Ctrl* emite un sunet (beep) şi afişează un mesaj care indică utilizatorului să apese tasta ENTER pentru a încheia programul. În programul *ctrlbrk.c*, funcţia returnează valoarea 1. Dacă programul returnează oricare altă valoare în afară de 0, programul va continua. Dacă programul returnează 0, programul se încheie.

776 UTILIZAREA SERVICIILOR DOS ÎN TRATAREA ERORILOR CRITICE



Atunci când DOS invocă un program de tratare a erorilor critice, trebuie să înţelegeţi că sistemul dumneavoastră este întrucâtva instabil – un serviciu al sistemului de operare a fost întrerupt brutal. În cadrul programului dumneavoastră de tratare a erorilor critice, ar trebui să restricţionaţi utilizarea serviciilor DOS la serviciile listate în tabelul 776.

Serviciu	Funcţie	Serviciu	Funcţie
01H	Intrare caracter	02H	Ieşire caracter
03H	Intrare port AUX	04H	Ieşire port AUX
05H	Ieşire imprimantă	06H	I/O directe la consolă
07H	Intrare caracter	08H	Intrare caracter
09H	Ieşire şir de caractere	0AH	Intrare tastatură în buffer
OBH	Testare stare de intrare	0CH	Golire buffer şi intrare
3300H	Obţine starea CTRL+C	3301H	Stabileşte starea CTRL+C
3305H	Obţine disc pornire	3306H	Obţine versiunea DOS
50H	Stabileşte PSP	51H	Obţine PSP
59H	Obţine eroarea extinsă	62H	Obţine PSP

Tabelul 776 *Serviciile DOS ce pot fi utilizate în tratarea erorilor critice.*

Dacă programele dumneavoastră trebuie să execute operaţii de I/O în tratarea erorilor critice, gândiţi-vă să utilizaţi funcţiile de I/O din fişierul antet *conio.h*.

CREȘTEREA PERFORMANȚEI PRIN UTILIZAREA SELECȚIEI SETULUI DE INSTRUCȚIUNI

C/C++ 777

În mod implicit, majoritatea compilatoarelor de mediu DOS generează programe care rulează pe orice sistem Intel, de la 8088 până la Pentium. Dacă știți dinainte că utilizatorul va rula un program numai pe o anumită mașină, puteți mări performanța programului prin utilizarea setului de instrucțiuni pentru cea mai avansată mașină. De exemplu, 80386 dispune de instrucțiuni care nu sunt posibile pe un 8088. Utilizând una dintre aceste instrucțiuni specifice pentru 80386, puteți înlocui câteva dintre instrucțiunile echivalente ale lui 8088. Însă, când beneficiați de avantajele acestor instrucțiuni, programele dumneavoastră nu vor mai putea rula pe mașini mai vechi. Pentru a genera un cod executabil pentru o anumită mașină, consultați opțiunile liniei de comandă a compilatorului dumneavoastră.

FUNCȚII INTRINSECI INLINE

C/C++ 778

Pentru creșterea performanțelor, multe compilatoare de C vă permit să înlocuiți funcțiile cu *cod inline*. În plus față de permisiunea de a utiliza cuvântul cheie *inline* înaintea funcției create, multe compilatoare de C vă permit să înlocuiți funcții de bibliotecă run-time cu funcții *inline* corespunzătoare. Funcțiile *intrinseci* pe care le puteți utiliza *inline* vor diferi de la un compilator la altul. Consultați documentația compilatorului dumneavoastră pentru a afla funcțiile disponibile. În cazul compilatorului Borland C++, puteți plasa *inline* funcțiile *intrinseci* listate în tabelul 778.

Funcții intrinseci pentru inline

alloc	fabs	*memchr*	*memcmp*	*memcpy*	*memset*	*rotl*
rotr	stpcpy	*strcat*	*strchr*	*strcmp*	*strcpy*	*strlen*
strncat	strncmp	*strncpy*	*strnset*	*strrchr*		

Tabelul 778 *Funcțiile intrinseci pentru **inline** acceptate de compilatorul de C.*

Pentru a instrui compilatorul să plaseze aceste funcții inline, puteți folosi un comutator al liniei de comandă sau funcția `#pragma intrinsic` prezentată în secțiunea 779.

ACTIVAREA ȘI DEZACTIVAREA FUNCȚIILOR INTRINSECI

C/C++ 779

În secțiunea 778 ați învățat că multe compilatoare de C vă permit înlocuirea anumitor funcții *intrinseci* cu cod *inline*. Utilizând opțiunile liniei de comandă a compilatorului, îi puteți indica acestuia să plaseze funcții *intrinsic inline*. În plus, multe preprocesoare acceptă funcția *pragma intrinsic*, care vă permite să activați sau să dezactivați funcțiile *intrinseci inline*:

```
#pragma intrinsic functie // activare inline
#pragma intrinsic -functie // dezactivare inline
```

Următoarea instrucțiune, de exemplu, cere compilatorului să genereze un cod inline pentru funcția *strlen*:

```
#pragma intrinsic strlen
```

Atunci când utilizați *pragma intrinsic*, trebuie să precedați *pragma* cu un prototip de funcție. Atunci când compilatorul întâlnește *pragma*, el înlocuiește numele funcției cu un nume echivalent care începe și se termină cu caracterul _. În cazul funcției *strlen*, compilatorul va genera constanta *_strlen_*:

```
#define strlen _strlen_
```

780 APELĂRILE RAPIDE DE FUNCȚII



Atunci când programul dumneavoastră invocă o funcție, compilatorul de C transmite parametrii către funcții prin stivă. Așa cum am arătat în capitolul despre funcții al acestei cărți, utilizarea stivei este responsabilă pentru majoritatea suprasarcinilor care corespund apelărilor de funcții. Pentru a face invocările de funcții să fie mai rapide, unele compilatoare de C dispun de modificatorul *_fastcall* pe care îl puteți plasa înaintea numelui funcției:

```
int _fastcall o_functie(int a, int b);
```

Următorul program, *fastcall.c*, ilustrează modificatorul *_fastcall*:

```
#include <stdio.h>
#include <time.h>
int _fastcall adun_rapid(int a, int b)
  {
    return(a + b);
  }
int adun_lent(int a, int b)
  {
    return(a + b);
  }
void main(void)
  {
    unsigned long int i, rezultat;
    clock_t start_time, stop_time;
    printf("Prelucreaza...\n");
    start_time = clock();
    for (i = 0; i < 2000000L; i++)
      rezultat = adun_rapid(i, -i);
    stop_time = clock();
    printf("Timpul de prelucrare pentru apelul rapid a fost de
           %d batai de ceas\n", stop_time - start_time);
    start_time = clock();
    for (i = 0; i < 2000000L; i++)
      rezultat = adun_lent(i, -i);
    stop_time = clock();
    printf("Timpul de prelucrare pentru apelul normal a fost de
           %d batai de ceas\n", stop_time - start_time);
  }
```

REGULI PENTRU TRANSMITEREA PARAMETRILOR _FASTCALL

C/C++ 781

În secțiunea 780 ați învățat că multe compilatoare acceptă funcția modificator *_fastcall*, care cere compilatorului să transmită parametri către funcție utilizând registrele. În funcție de calculator, numărul de registre disponibile pentru parametri va fi diferit. În cazul compilatorului Borland C++, programele dumneavoastră pot transmite numai trei parametri prin intermediul registrelor. Tabelul 781 specifică modul în care modificatorul *_fastcall* transmite parametrii funcțiilor atunci când este utilizat cu compilatorul Borland C++.

Tip parametru	Registre utilizate
char (signed și unsigned)	AL, DL, BL
int (signed și unsigned)	AX, DX, BX
long (signed și unsigned)	DX, AX
pointer *near*	AX, DX, BX
altele	Transmiși în stivă

Tabelul 781 *Registre utilizate pentru transmiterea parametrilor cu modificatorul **_fastcall**.*

CODUL INVARIANT

C/C++ 782

Examinând directivele compilatorului care influențează optimizarea, puteți întâlni termenul *cod invariant*. În general, termenul de cod invariant se referă la instrucțiuni care apar în cadrul unei bucle ale cărei valori nu se modifică. De exemplu, următoarea buclă *for* atribuie rezultatul înmulțirii *a*b*c* fiecăruia dintre elementele matricei:

```
for (i = 0; i < 100; i++)
    matrice[i] = a * b * c;
```

Din cauză că variabilele a, b și c nu se modifică pe parcursul buclei, rezultatul înmulțirii este *invariant* (nu se modifică). Atunci când programați, trebuie să fiți atenți la codul invariant. Când întâlniți cod invariant, puteți crește de obicei performanțele programului, modificându-l. În cazul buclei precedente *for*, puteți crește performanțele programului înlocuind înmulțirea cu rezultatul său, ca mai jos:

```
rezultat = a * b * c;
for (i = 0; i < 100; i++)
    matrice[i] = rezultat;
```

Pentru a mări performanțele programului, majoritatea compilatoarelor vor verifica apariția codului invariant, înlocuindu-l în cadrul codului obiect destinație cu echivalentul său non-invariant. Ideal ar fi, însă, să găsiți chiar dumneavoastră codul invariant și să-l corectați. Totuși, utilizând opțiuni în linia de comandă, puteți cere compilatorului dumneavoastră să execute aceste substituiri pentru dumneavoastră, în timpul compilării.

ELIMINAREA ÎNCĂRCĂRII REDUNDANTE

C/C++ 783

După cum ați învățat, pentru a mări performanța, compilatorul de C încarcă adeseori valorile în registre. Atunci când compilatorul efectuează *o eliminare a încărcării redundante*,

compilatorul urmăreşte valorile pe care le-a plasat deja în registre. Compilatorul se adresează apoi registrelor când necesită o valoare, în loc să o reîncarce pentru a doua oară. Compilatorul utilizează suprimarea încărcării redundante pentru a preveni duplicarea operaţiilor de încărcare, mărind astfel performanţele programului. Dezavantajul utilizării eliminării încărcării redundante este acela că va dura puţin mai mult compilarea programelor dumneavoastră. De regulă, însă, ar trebui să indicaţi întotdeauna compilatorului să efectueze eliminarea încărcării.

784 COMPACTAREA CODULUI

C/C++

Atunci când examinaţi documentaţia compilatorului, veţi întâlni termenul *compactare de cod*. În general, compactarea de cod utilizează ramificaţii la codul anterior pentru eliminarea instrucţiunilor redundante. De exemplu, să analizăm următorul program, *compact.c*:

```
#include <stdio.h>

void main(void)
{
    int a = 1, b, c, d;
    switch (a) {
    case 1:  a = 5;
             b = 6;
             c = 7;
             d = 8;
          break;
    case 2:  b = 6;
             c = 7;
             d = 8;
          break;
    };
}
```

Dacă examinaţi instrucţiunea *switch*, veţi constata că instrucţiunile executate de program pentru fiecare *case* sunt foarte asemănătoare. În loc să dubleze instrucţiunile de atribuire în ambele locaţii, compilatorul poate plasa o instrucţiune JMP (de la *jump* care în limbajul de asamblare este echivalent cu *goto*), care execută un salt înapoi la instrucţiunea *b = 6*, care apare în primul *case* şi la începutul celui de al doilea.

785 COMPACTAREA BUCLEI

C/C++

Dacă examinaţi buclele *for* care apar în programele dumneavoastră, veţi observa probabil că majoritatea buclelor operează cu un şir de caractere sau cu alte matrice. Atunci când programele dumneavoastră atribuie aceeaşi valoare fiecărui element dintr-o matrice, compilatorul de C poate optimiza performanţa programului dumneavoastră prin înlocuirea buclei cu una dintre instrucţiunile *80x86 STxxx*. De exemplu, următoarea buclă *for* iniţializează matricea *sir_null* la *NULL*:

```
for (i = 0; i < sizeof(sir_null); i++)
     sir_null[i] = NULL;
```

Dacă veți examina ieșirea în limbaj de asamblare produsă de compilator, veți observa că bucla a fost eliminată de compilator, utilizând instrucțiunea *STOSW*. Astfel de substituiri ale compilatorului sunt denumite *compactări de bucle*.

INDUCȚIA BUCLEI ȘI REDUCȚIA PUTERII

C/C++ 786

Inducția buclei și reducția puterii sunt tehnici pe care compilatorul le utilizează pentru optimizarea buclelor din cadrul unui program. De obicei, compilatorul optimizează buclele în cadrul unui program atunci când programul operează cu matrice în interiorul buclei. De exemplu, să analizăm următoarea buclă, care atribuie valori elementelor unei matrice:

```
for (i = 0; i < 128; i++)
    matrice[i] = 0;
```

Pentru fiecare referință la matrice, compilatorul trebuie să execute o operație de înmulțire pentru a determina elementul corect (*baza + i * sizeof(tip_matrice)*). În loc să utilizeze matricea, compilatorul poate utiliza un pointer:

```
end = &matrice[128];
for (ptr = matrice; ptr < end; ptr++)
    *ptr =0;
```

Prin eliminarea acestei înmulțiri lente, compilatorul mărește performanța programului. Procesul de creare a noilor variabile din variabilele buclei este denumit *inducția buclei*. Deoarece variabilele induse sunt de obicei mai puțin complexe decât variabilele pe care le înlocuiesc, bucla nou creată introduce o *reducție puterii*.

ELIMINAREA SUBEXPRESIILOR COMUNE

C/C++ 787

Dacă programul dumneavoastră lucrează cu matrice, uneori puteți îmbunătăți performanțele programului prin eliminarea subexpresiilor comune. De exemplu, să luăm următoarea instrucțiune *if*, care testează dacă un element dintr-o matrice conține litera A majusculă sau minusculă:

```
if ((matrice[i] == 'A') || (matrice[i] == 'a'))
```

Pentru fiecare test, compilatorul trebuie să rezolve fiecare element al matricei efectuând o înmulțire (*baza + i * sizeof(tip_matrice))*. O implementare mai rapidă însă, ar înlocui referințele la matrice cu un pointer:

```
ptr = &matrice[i];
if ((*ptr == 'A') || (*ptr == 'a'))
```

Codul alternativ, în care subexpresiile comune au fost înlocuite cu altele mai rapide, mărește performanța programului. În anumite cazuri, însă, încercarea de a elimina subexpresiile comune poate face programul dumneavoastră mai dificil de înțeles. Multe compilatoare de C vor găsi avantajoasă eliminarea subexpresiilor comune și vor efectua operațiile pentru dumneavoastră în cadrul codului rezultant (compilat). Când lucrați cu condiții complexe, gândiți-vă că puteți crește performanța programului dumneavoastră prin eliminarea sau reducerea subexpresiilor comune.

788 CONVERSIILE STANDARD C



Atunci când efectuați operații aritmetice cu diferite tipuri de valori, compilatorul de C *promovează* adesea tipul celei mai mici valori. Pentru a vă ajuta să înțelegeți conversiile standard din C, parcurgeți următoarele reguli, pe care compilatorul de C le aplică în ordine, de la prima la ultima.

- Cu excepția valorilor de tipul *unsigned short*, compilatorul de C promovează toate valorile întregi mici la *int*. Compilatorul de C promovează valorile de tip *unsigned short* la *unsigned int*.
- Dacă unul dintre operanzi este *long double*, compilatorul de C promovează pe celălalt la *long double*.
- Dacă unul dintre operanzi este *double*, compilatorul de C promovează pe celălalt la *double*.
- Dacă unul dintre operanzi este *float*, compilatorul de C promovează pe celălalt la *float*.
- Dacă unul dintre operanzi este *unsigned long*, compilatorul de C promovează pe celălalt la *unsigned long*.
- Dacă unul dintre operanzi este *long*, compilatorul de C promovează pe celălalt la *long*.
- Dacă unul dintre operanzi este *unsigned*, compilatorul de C promovează pe celălalt la *unsigned*.
- În celelalte cazuri, compilatorul de C tratează ambii operanzi ca fiind de tip *int*.

789 CELE PATRU TIPURI DE BAZĂ ALE LIMBAJULUI C



Dacă studiați declarațiile complexe din C, rețineți că limbajul C acceptă patru tipuri de bază: *void, scalar, function* și *aggregate*. Tipul *void* specifică absența valorilor. De exemplu, *void* într-o listă de parametri indică faptul că funcția nu trebuie să primească nici un parametru. De asemenea, *void* înaintea unui nume de funcție specifică faptul că funcția nu returnează o valoare. Valorile *scalare* includ valori aritmetice, enumerări, pointeri și referințe. Tipul *function* specifică o funcție care returnează un anumit tip. În sfârșit, un tip *aggregate* specifică o matrice, uniune, structură sau clasă C++. Dacă studiați declarațiile complexe, încercați să mapați declarația la unul dintre tipurile acceptate de C.

790 TIPURILE FUNDAMENTALE FAȚĂ DE TIPURILE DERIVATE



Atunci când examinați declarațiile complexe în C, nu uitați că limbajul C acceptă tipuri *fundamentale* și tipuri *derivate*. Tipurile fundamentale de date sunt următoarele: *void, char, double, float* și *int*. În plus, limbajul C permite aplicarea modificatorilor *long, short, signed* și *unsigned* la tipurile fundamentale. Tipurile derivate, pe de altă parte, cuprind: matricele, clasele, funcțiile, pointerii, structurile și uniunile de alte tipuri. Tipurile *char, int, long și short* sunt tipuri întregi. Dacă examinați declarațiile, următoarele tipuri întregi vor fi echivalente:

```
char, signed char // tip implicit dat de compilator
int, signed int
unsigned, unsigned int
short, short int, signed short int
unsigned short, unsigned short int
long, long int, signed long int
unsigned long, unsigned long int
```

INIȚIALIZATORII

C/C++ 791

Inițializatorii sunt valorile pe care programele dumneavoastră le atribuie variabilelor la declarare. Când utilizați inițializatori, țineți seama de următoarele reguli:

- Dacă programul nu inițializează explicit un tip aritmetic, majoritatea compilatoarelor vor inițializa variabila cu 0.
- Dacă programul nu inițializează explicit un tip pointer, majoritatea compilatoarelor vor inițializa variabila cu *NULL*.
- Dacă numărul inițializatorilor depășește numărul variabilelor care urmează a fi inițializate, compilatorul va genera o eroare.
- Toate expresiile utilizate la inițializarea variabilelor, trebuie să fie constante (regula nu se aplică și la C++), în cazul în care inițializatorii atribuie obiecte statice, matrice, structuri sau uniuni.
- Dacă variabila declarată are valabilitate de bloc, iar programul nu a declarat variabila ca fiind externă, declararea nu poate avea inițializatori.
- Când codul program pune la dispoziție mai puțini inițializatori decât cei solicitați de compilator pentru inițializarea completă a variabilelor, compilatorul va inițializa restul de valori conform unei tehnici de inițializare implicită.

EDITORUL DE LEGĂTURI

C/C++ 792

Așa cum ați învățat, editorul de legături combină codul din programe, fișiere obiect și biblioteci. În funcție de editorul de legături utilizat în cadrul compilării, uneori probabil că două sau mai multe funcții din fișierele legate vor avea același nume. *Legarea* este procesul de stabilire a funcției pe care editorul de legături o aplică unei referințe. În limbajul C, identificatorii au un singur atribut de legare din cele trei posibile: *extern, intern* și *fără legare*. Identificatorul cu legare *externă* reprezintă același obiect în toate fișierele care alcătuiesc programul. Identificatorul cu legare *internă* reprezintă același obiect în cadrul unui fișier. Identificatorul fără legare este unic pentru toate fișierele – însemnând că apare o singură dată. Editorul de legături utilizează tipul de legare al identificatorului pentru a determina care funcție C se asociază identificatorului. Editorul de legături utilizează următoarele reguli de legare:

- Identificatorii declarați ca *statici* au *legare internă*.
- Dacă un identificator apare cu legare internă și externă, C va utiliza legarea internă, iar C++ va utiliza legarea externă.

- Dacă programul declară un identificator cu cuvântul cheie *extern*, identificatorul are aceeaşi legare ca orice declaraţie vizibilă în aria de valabilitate în fişier; dacă o astfel de declaraţie nu există, identificatorul este legat extern.
- Dacă identificatorul unei funcţii nu deţine un specificator de clasă de stocare, identificatorul are aceeaşi legare ca şi când programul ar utiliza cuvântul cheie *extern* asociat identificatorului.
- Identificatorii de obiect declaraţi fără specificator de clasă de stocare au legare externă.
- Identificatorii pe care îi declaraţi ca altceva decât un obiect sau funcţie nu au legare.
- Parametrii funcţiilor nu au legare.
- Identificatorii care au valabilitate de bloc declaraţi fără clasa de stocare *extern* nu au legătură.

793 DECLARAŢII DE PROBĂ



Declaraţia de probă este o declaraţie de date externe care nu are specificatori de clasă de stocare şi nici o iniţializare. De exemplu, să presupunem că un compilator întâlneşte următoarea declaraţie:

```
int valoare;
```

Dacă, mai târziu, compilatorul întâlneşte o definiţie a variabilei, compilatorul consideră variabila ca şi cum ar fi precedată de cuvântul cheie *extern*. Când compilatorul ajunge la sfârşitul unităţii de conversie, fără să întâlnească o definiţie, compilatorul alocă memorie pentru variabilă. Următorul program, *tentative.c*, formulează o declaraţie de variabilă de probă pentru variabila *valoare*:

```
#include <stdio.h>

int valoare;
void main(void)
  {
    printf("%d\n", valoare);
  }
int valoare = 1500;
```

Când compilatorul întâlneşte definiţia variabilei valoare, care iniţializează variabila cu 1500, compilatorul va converti prima declaraţie a variabilei *valoare* dintr-o declaraţie de probă, într-o definire deplină.

794 DEOSEBIREA DINTRE DECLARAŢII ŞI DEFINIŢII



Multe dintre secţiunile prezentate de-a lungul acestei cărţi utilizează termenii „declaraţie" şi „definiţie". În general, o declaraţie introduce într-un program unul sau mai mulţi identificatori. Pe de altă parte, o definiţie cere compilatorului să aloce efectiv memorie pentru obiect. De exemplu, puteţi considera prototipul unei funcţii drept declaraţie, iar antetul funcţiei, ca definiţie. Limbajul C clasifică declaraţiile ca *declaraţii de definire* sau ca *declaraţii de*

referențiere. O declarație de definire declară unul sau mai mulți identificatori și definește cantitatea de memorie pe care compilatorul urmează să o aloce obiectului. O declarație de *referențiere* introduce pur și simplu un identificator. În limbajul C, pot fi declarate obiectele listate în tabelul 794.

Obiecte			
Matrice	Clase	Membri de clasă	Enumerări
Etichete enumerate	Constante	Funcții	Etichete
Macro	Structuri	Membri de structuri	Tipuri
Uniuni	Membri de uniuni	Variabile	

Tabelul 794 *Tipuri declarabile în limbajul C.*

VALORILE L (LVALUE)

C/C++ 795

Atunci când programele dumneavoastră lucrează cu pointeri, este posibil să întâlniți mesaje de erori la compilare care afirmă că se solicită o valoare *l (l-left-stânga).* O valoare *l* este o expresie pe care compilatorul o poate utiliza pentru localizarea unui obiect. Puteți considera valorile *l* ca expresii care ar fi valide în partea stângă a operatorului de atribuire. Următoarele exemple sunt valori *l* valide:

```
variabila = valoare;
*variabila = valoare;
variabila[i] = valoare;
```

Este important de remarcat, totuși, că fiecare dintre valorile *l* prezentate mai sus ar fi putut fi la fel de bine să fie plasate și în partea dreaptă a operatorului de atribuire. O valoare *l* pur și simplu pune la dispoziție o valoare pe care compilatorul o poate utiliza pentru localizarea unui obiect în memorie. Limbajul C acceptă valori *l modificabile* și *nemodificabile.* O valoare *l modificabilă* este o valoare pointer pe care o puteți modifica pentru a indica diferite valori. Majoritatea pointerilor pe care i-ați utilizat în cadrul acestei cărți sunt valori *l modificabile.* Pe de altă parte, valorile *l nemodificabile* nu pot fi modificate. O constantă pointer, de exemplu, este o valoare *l nemodificabilă.*

VALORILE R (RVALUE)

C/C++ 796

Atunci când compilați un program, este posibil să întâlniți un mesaj de eroare la compilare care afirmă că s-a întâlnit o valoare *r* (r-right-dreapta) pe care nu o aștepta. O valoare *r* este o expresie care apare în partea dreaptă a unui semn egal. Următoarele expresii sunt exemple de valori *r*:

```
rezultat = valoare;
rezultat = 1001;
rezultat = valoare + 1500;
```

Următoarele instrucțiuni, însă, sunt nevalide deoarece nu specifică locația de memorie la care compilatorul de C poate atribui o valoare:

```
1500 = rezultat;
valoare + 1500 = rezultat;
```

Astfel de declarații eronate apar de obicei, atunci când utilizatorul încearcă să creeze un pointer:

```
*(valoare + 1500) = rezultat;
```

797 UTILIZAREA CUVINTELOR CHEIE PENTRU REGISTRELE SEGMENT

C/C++

După cum ați învățat, PC-ul utilizează patru registre specifice pentru a localiza codul programului, datele și stiva. Programele dumneavoastră pot utiliza funcția de bibliotecă run-time *segread* pentru a determina valoarea registrelor. În plus, multe compilatoare de mediu DOS dispun de cuvintele cheie listate în tabelul 797.

Cuvânt cheie	Semnificație
_cs	Creează un pointer către segmentul cod
_ds	Creează un pointer către segmentul de date
_es	Creează un pointer către segmentul suplimentar
_ss	Creează un pointer către segmentul stivei

Tabelul 797 *Cuvintele cheie suplimentar pentru registre puse la dispoziție de multe compilatoare de mediu DOS.*

Următoarea declarare creează un pointer către segmentul de stivă:

```
char _ss *segm_stiva;
```

În funcție de modelul de memorie curent, pointerii vor conține pointeri *far* sau *near*, după caz.

798 UTILIZAȚI CU GRIJĂ POINTERII FAR

C/C++

În capitolul despre pointeri al acestei cărți, ați învățat că un pointer *far* este un pointer de 32 de biți care conține o adresă de segment de 16 biți și o adresă de deplasament de 16 biți. Pointerii *far* permit programelor dumneavoastră să acceseze intervalul de 1Mb al memoriei convenționale a PC-ului. Atunci când utilizați pointeri *far* însă, trebuie să înțelegeți modul în care valoarea pointerului se ajustează când valoarea deplasamentului depășește limita de 16 biți. Să presupunem, de exemplu, că pointerul *locatie* de tip *far char* conține următoarea valoare:

```
locatie = 0x1000FFFE; // Segment 0x1000 Deplasament FFFE
```

Dacă incrementați pointerul, valoarea pointerului va arăta astfel:

```
locatie = 0x1000FFFF; // Segment 0x1000 Deplasament FFFF
```

Dacă incrementați din nou pointerul, rezultatul de eroare posibil va arăta astfel:

```
locatie = 0x10000000; // Segment 0x1000 Deplasament 0000
```

Observați că valoarea deplasamentului s-a ajustat la zero, dar valoarea segment nu s-a modificat. Ca rezultat al procesului de incrementare, pointerul s-a ajustat la începutul adresei

segmentului de 64Kb. Dacă este nevoie ca pointerul să se deplaseze la începutul următorului segment, utilizaţi un pointer *huge*.

POINTERII NORMALIZAŢI

C/C++799

După cum ştiţi, PC-ul adresează locaţiile de memorie utilizând adrese de segment şi de deplasament. PC-ul acceptă până la 65636 adrese de segment. Fiecare adresă de segment începe la adresa de 16 octeţi: 0, 16, 32, 48 şi aşa mai departe. Dacă înmulţiţi adresa de 16 octeţi cu 65636 segmente, rezultă un spaţiu de adresare de 1Mb. Fiind dată o adresă de segment, adresa de deplasament permite alegerea uneia din cele 65636 locaţii posibile din cadrul segmentului. Când utilizaţi adrese de segment şi de deplasament, puteţi adresa fiecare locaţie din memorie folosind combinaţii diferite de segmente şi deplasamente. Să presupunem, de exemplu, că vreţi să adresaţi locaţia 48 din memorie. Pentru aceasta, puteţi utiliza oricare din următoarele combinaţii segment/deplasament:

- Segment 0 Deplasament 48
- Segment 1 Deplasament 32
- Segment 2 Deplasament 16
- Segment 3 Deplasament 0

Deoarece puteţi face referinţă la fiecare locaţie de memorie în mai multe feluri, este posibil ca doi pointeri *far* să facă referinţă la aceeaşi locaţie de memorie, dar să conţină valori diferite. Analizaţi următoarele alocări de pointer:

```
char far *ptr1 = 0x00000030; // Segment 0 Deplasament 48
char far *ptr2 = 0x00030000; // Segment 3 Deplasament 0
```

Ambii pointeri vor face referire la aceeaşi locaţie de memorie. Însă, dacă programul compară valorile pointerilor, ele nu vor fi egale. Un pointer normalizat elimină acest decalaj, prin permanenta stocare a valorilor, în aşa fel încât compilatorul să utilizeze un deplasament de 16 octeţi. În acest mod, compilatorul stochează întotdeauna adresele de segment utilizând segmentul cel mai apropiat de valoare. În cazul precedent, pointerul normalizat conţine segment 3 şi deplasament 0.

INSTRUCŢIUNILE COPROCESORULUI MATEMATIC

C/C++800

Un procesor în virgulă mobilă (sau matematic) este un cip specializat care conţine instrucţiuni care efectuează rapid operaţii matematice – cum sunt împărţirea, înmulţirea şi chiar calculele de extragere a rădăcinii pătrate – utilizând valori în virgulă mobilă. Dacă utilizaţi un procesor 8088, 80286 sau 80386, trebuie să procuraţi un coprocesor în virgulă mobilă, de tip 8087, 80287 sau 80387. Dacă utilizaţi un calculator 80486DX sau mai avansat, procesorul în virgulă mobilă este construit în procesorul principal. Deoarece coprocesoarele matematice efectuează numai operaţii în virgulă mobilă, ele pot efectua operaţii rapide. Când calculatorul dumneavoastră posedă un coprocesor matematic, trebuie să indicaţi calculatorului să genereze instrucţiuni care utilizează coprocesorul. În funcţie de compilatorul folosit, opţiunile care trebuie utilizate pentru a genera aceste instrucţiuni pentru coprocesorul în virgulă mobilă vor diferi. Pentru a permite programelor să ruleze pe sisteme

care nu posedă coprocesor matematic, majoritatea compilatoarelor nu generează în mod implicit instrucțiuni în virgulă mobilă. Dacă utilizați *Turbo C++ Lite*, compilatorul va accepta numeroase opțiuni */FPx* care vă permit să indicați compilatorului să utilizeze întotdeauna instrucțiuni în virgulă mobilă. Opțiunile pentru virgulă mobilă pe care le selectați vor afecta dimensiunea și viteza programului. Consultați documentația compilatorului dumneavoastră pentru informații suplimentare privind opțiunile de virgulă mobilă ale compilatorului.

801 DECLARAȚIILE DE VARIABILE CU CDECL ȘI PASCAL

C/C++

Examinând programe care utilizează module cu mai multe limbaje, cum sunt Pascal sau C, veți întâlni variabile declarate cu modificatorii *pascal* și *cdecl*. Pentru a păstra compatibilitatea cu identificatorii din Pascal, modificatorul *pascal* cere compilatorului să nu țină seama de diferența dintre literele mari și mici și să nu preceadă identificatorul cu liniuța de subliniere. Modificatorul *cdecl*, pe de altă parte, cere compilatorului să asigure diferențierea dintre literele mari și mici și includerea liniuțelor de subliniere. Următorul program declară o variabilă externă denumită *numar* definită într-un program Pascal:

```
#include<stdio.h>

extern int pascal numar;
void main(void)
   {
     printf("Valoarea este %d\n", numar);
}
```

802 PREVENIREA DIRECTIVELOR INCLUDE CIRCULARE

C/C++

Pe măsură ce programele utilizează din ce în ce mai mult fișiere antet (programele C++ definesc de multe ori clasele în fișierele antet), vor exista situații în care un fișier inclus va include un al doilea fișier antet care, la rândul lui, va include primul fișier antet. Pe măsură ce preprocesorul efectuează includerile, se poate ajunge la situația unor operații circulare. Pentru a reduce posibilitatea operațiilor circulare, fișierele antet pot declara o macrocomandă care va preveni compilatorul să proceseze fișierele pentru a doua oară. De exemplu, următorul fișier antet utilizează macrocomanda *GRUP_DEFINIT* pentru a determina dacă s-a procesat deja conținutul:

```
#ifndef GRUP_DEFINIT
#define GRUP_DEFINIT 1

        // Alte instructiuni include
#endif
```

În acest caz, când compilatorul procesează pentru prima dată fișierul antet, el definește o macrocomandă. Dacă programul include fișierul antet a doua oară, compilatorul nu va procesa conținutul fișierului, datorită directivei *ifndef*.

INTRODUCERE ÎN C++

C/C++ 803

C++ este un limbaj de programare dezvoltat de dr. Bjarne Stroustrup în laboratoarele AT&T Bell care are la bază limbajul de programare C la care a adăugat orientarea pe obiecte și alte facilități. Puteți considera limbajul C++ ca pe un supraset al lui C, deoarece C++ acceptă caracteristicile limbajului de programare C pe care le-ați învățat de-a lungul acestei cărți. Așa cum veți învăța, C++ este mai mult decât „un C orientat pe obiecte" – C++ adaugă, de fapt, multe noi caracteristici care sporesc capacitățile programelor dumneavoastră. Dacă utilizați un compilator de C++ (cum este *Turbo C++ Lite*), majoritatea programelor prezentate în precedentele 802 secțiuni vor fi compilate și executate cu succes, fără a fi necesare modificări. Secțiunile următoare încep cu bazele limbajului C++ și se bazează pe cunoștințele dumneavoastră despre limbajul C. Atunci când veți ajunge la finalul acestei cărți, veți fi bine pregătit în C și C++.

CUM DIFERĂ FIȘIERELE SURSĂ DIN C++

C/C++ 804

În general, nu există diferențe între fișierele sursă din C și C++. Ambele limbaje acceptă pe deplin directivele de compilator, cum ar fi *#include* și *#define*. În ceea ce privește denumirea, mulți programatori utilizează extensia CPP pentru a diferenția fișierele sursă C++ de fișierele sursă în C. Tot ce ați învățat în prima parte a acestei cărți se aplică și la crearea programelor în C++, cu excepția unor structuri și constante unice. Adică, puteți include în C++ fișierele antet, bibliotecile de legături ale codului obiect și așa mai departe.

UN EXEMPLU SIMPLU DE PROGRAM ÎN C++

C/C++ 805

În secțiunea 2, ați creat primul dumneavoastră program în C, care utiliza *printf* pentru a afișa un mesaj pe ecran:

```
#include <stdio.h>

void main(void)
  {
    printf("Totul despre C/C++");
  }
```

Următorul program în C++, *simplu.cpp*, execută un proces identic:

```
#include <iostream.h>

void main(void)
  {
    cout << "Totul despre C/C++";
  }
```

Programul *simplu.cpp* utilizează fluxul I/O de C++ *cout*, prezentat în secțiunea 806. Puteți compila și construi programul *simplu.cpp* la fel cum ați construit toate programele în C de până acum. Atunci când veți executa programul *simplu.cpp*, pe ecran se va afișa următorul mesaj:

```
Totul despre C/C++
C:\>
```

***Observație:** După cum vedeți, codul C++ nu utilizează caracterul **escape** \' pentru apostroful din cadrul șirului de caractere la ieșire. Majoritatea compilatoarelor de C++ nu apelează la caracterul **escape** \' și de aceea, în celelalte șiruri de caractere din această carte nu se va mai utiliza notarea \'.*

806 FLUXUL DE INTRĂRI/IEȘIRI COUT

C/C++

În secțiunea 805, exemplul de program utilizează fluxul I/O *cout* pentru a scrie un șir de caractere pe ecran:

```
cout << "Totul despre C/C++";
```

Redirectarea ieșirii la fluxul I/O *cout* este aceeași ca în cazul utilizării instrucțiunii *printf* pentru a scrie ieșirea către *stdout*. Dublul simbol „mai mic decât" (<<) nu este un operator pe biți de deplasare la stânga. În schimb, simbolul este un operator de ieșire care specifică fluxul la care programul transmite datele. Următorul program, *cout.cpp*, utilizează operatorul C++ de ieșire pentru a afișa câteva mesaje diferite:

```
#include <iostream.h>

void main(void)
  {
    cout << "Aceasta este linia unu.\n";
    cout << "Acest text este pe ";
    cout << "linia a doua.\n";
    cout << "Aceasta este ultima linie.";
  }
```

Atunci când compilați și executați programul *cout.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este linia unu.
Acest text este pe linia a doua.
Aceasta este ultima linie.
C:\>
```

807 SCRIEREA VALORILOR ȘI VARIABILELOR CU COUT

C/C++

Așa cum ați învățat, fluxul de ieșire *cout* permite programelor dumneavoastră afișarea rezultatului pe ecran. Secțiunile precedente au utilizat *cout* pentru a afișa șiruri de caractere. Următorul program, *cout_num.cpp*, utilizează *cout* pentru a afișa șiruri de caractere și numere:

```
#include <iostream.h>

void main(void)
  {
cout << "cout va permite afisarea de siruri de caractere,
  valori int si float\n";
    cout << 1500;
    cout << "\n";
```

```
    cout << 1.2345;
  }
```

Atunci când compilaţi şi executaţi programul *cout_num.cpp*, ecranul dumneavoastră va afişa următorul rezultat:

```
cout va permite afisarea de siruri de caractere, valori int si float
1500
1.2345
C:\>
```

COMBINAREA DIFERITELOR TIPURI DE VALORI CU COUT

C/C++ 808

În secţiunea 807 aţi învăţat că fluxul I/O *cout* permite programelor dumneavoastră să afişeze toate tipurile de valori. Programul *cout_num.cpp* prezentat în secţiunea 807, utilizează câteva instrucţiuni pentru a afişa ieşirile sale:

```
cout << "cout va permite afisarea de siruri de caractere,
  valori int si float\n";
    cout << 1500;
    cout << "\n";
    cout << 1.2345;
```

Din fericire, *cout* permite plasarea diferitelor tipuri de valori în fluxul de ieşire într-o singură instrucţiune, cum arătăm în următorul program, *cout_una.cpp*:

```
#include <iostream.h>

void main(void)
  {
    cout << "cout afiseaza siruri de caractere " << 1500
      << "\n" << 1.2345;
  }
```

AFIŞAREA VALORILOR HEXAZECIMALE ŞI OCTALE

C/C++ 809

Aşa cum aţi învăţat, fluxul I/O *cout* permite programelor dumneavoastră afişarea valorilor de tip *int* şi *float*. Atunci când utilizaţi *printf* pentru afişarea valorilor întregi, puteţi utiliza specificatorii de format *%x* şi *%o* pentru a afişa valorile în hexazecimal şi octal. Când programele dumneavoastră utilizează *cout* pentru afişarea ieşirii, puteţi utiliza modificatorii *dec*, *oct* şi *hex*, cum arătăm în următorul program, *cout_hex.cpp*:

```
#include <iostream.h>

void main(void)
  {
    cout << "Valoare zecimala " << dec << 0xFF;
    cout << "\nValoare octala " << oct << 10;
```

```
    cout << "\nValoare hexazecimala " << hex << 255;
  }
```

810 REDIRECTAREA IEȘIRII

C/C++

După cum ați învățat, programele dumneavoastră pot utiliza fluxul I/O *cout* pentru a afișa ieșirea, ca și cum ați fi scris ieșirea direct către *stdout*. De aceea, dacă aveți un program care utilizează *cout*, puteți redirecta ieșirea lui către un fișier sau către un alt dispozitiv. Următorul program, *1_la_100.cpp*, utilizează *cout* pentru a afișa numerele de la 1 la 100:

```
#include <iostream.h>

void main(void)
  {
    int i;
    for (i = 1; i <= 100; i++)
      cout << i << '\n';
  }
```

Utilizând operatorul DOS de redirectare a ieșirii, puteți să redirectați ieșirea programului către un fișier:

```
C:\> 1_LA_100 >> NUMEFIS.EXT <ENTER>
```

811 DACĂ PREFERAȚI PRINTF, UTILIZAȚI-L

C/C++

Câteva dintre secțiunile precedente au executat ieșirea utilizând fluxul I/O *cout*. Dacă vi se pare mai convenabil să utilizați funcția *printf*, utilizați *printf*. În secțiunile care urmează, veți învăța cum se formatează mai bine ieșirea afișată de programele dumneavoastră cu *cout*. Din acest moment, puteți alege utilizarea lui *cout* pentru toate ieșirile programelor dumneavoastră. De regulă, însă, pentru ca programele dumneavoastră să fie mai ușor de înțeles, ar trebui să alegeți una din tehnici și să rămâneți la ea. Următorul program, *ambele.cpp*, afișează ieșirea utilizând atât *cout*, cât și *printf*:

```
#include <iostream.h>
#include <stdio.h>

void main(void)
  {
    cout << "Totul ";
    printf("despre ");
    cout << "C/C++";
  }
```

812 SCRIEREA IEȘIRII LA CERR

C/C++

După cum știți, atunci când programele dumneavoastră scriu ieșirea la indicatorul de fișier *stderr*, C++ nu poate redirecta ieșirea de la ecran. Dacă utilizați fluxuri I/O ale limbajului C++

pentru a executa intrări și ieșiri, programele dumneavoastră pot scrie la fluxul I/O *cerr*. Următorul program, *utilcerr.cpp*, utilizează *cerr* pentru a preveni ca utilizatorul sau programul să redirecteze ieșirea:

```
#include <iostream.h>

void main(void)
  {
    int i;
    for (i = 1; i <= 100; i++)
      cerr << "Nu se poate redirecta cerr " << i << '\n';
  }
```

INTRĂRILE PRIN CIN

C/C++813

Așa cum ați învățat, fluxul I/O *cout* permite programelor dumneavoastră afișarea către *stdout*. În mod similar, programele dumneavoastră în C++ pot primi intrarea utilizând fluxul I/O *cin*. Următorul program, *util_cin.cpp*, utilizează *cin* pentru a primi intrări pentru diferite tipuri de variabile:

```
#include <iostream.h>

void main(void)
  {
    int varsta;
    float salariu;
    char nume[128];
    cout << "Introduceti numele dumneavoastra varsta salariul: ";
    cin >> nume >> varsta >> salariu;
    cout << nume << ' ' << varsta << ' ' << salariu;
  }
```

FLUXUL CIN NU UTILIZEAZĂ POINTERI

C/C++814

Cum ați aflat în secțiunea 813, fluxul *cin* nu utilizează pointeri pentru referința la variabile. După cum ați învățat, când primiți informațiile de intrare cu o funcție cum ar fi *scanf*, transmiteți explicit un pointer la variabila care primește informația (astfel că funcția poate modifica variabila). Însă, datorită construcției fluxului *cin*, nu trebuie să transmiteți un pointer la variabilă pentru *cin* – dacă o faceți, *cin* va returna o eroare. De exemplu, următorul program, *cin_er.cpp*, va returna șase erori atunci când încercați să compilați programul:

```
#include <iostream.h>
void main(void)
  {
    int varsta;
    float salariu;
    char nume[128];
```

```
    cout << "Introduceti numele varsta salariul: ";
    cin >> &nume >> &varsta >> &salariu;
    cout << nume << ' ' << varsta << ' ' << salariu;
}
```

Veți învăța în secțiunile următoare modul în care *cin* plasează valorile în cadrul variabilelor care nu sunt pointeri. Pentru moment, trebuie să înțelegeți numai că programele dumneavoastră transmit către *cin* numele real al variabilei.

815 CUM SELECTEAZĂ CIN CÂMPURILE DE DATE

C/C++

În secțiunea 813, ați utilizat fluxul I/O *cin* pentru a citi numele utilizatorului, vârsta și salariul pe o linie:

```
cin >> nume >> varsta >> salariu;
```

Atunci când programele dumneavoastră utilizează *cin* pentru a citi intrarea, trebuie să înțelegeți modul în care *cin* analizează intrarea. Menționăm că *cin* utilizează spații albe (spațiu, tab sau linie nouă) pentru a delimita câmpurile. De aceea, dacă utilizatorul tipărește numele său întreg (cum ar fi Ion Ionescu) în operația precedentă, *cin* ar trebui să folosească primul nume pentru variabila *nume* și al doilea pentru variabila *varsta*, iar operația de I/O ar fi eronată. În secțiunile care urmează, veți învăța cum se execută o intrare formatată la utilizarea lui *cin*.

816 MODUL ÎN CARE FLUXURILE I/O RECUNOSC TIPURILE VALORILOR

C/C++

După cum ați învățat, programele dumneavoastră pot utiliza fluxurile I/O *cin*, *cout* și *cerr* pentru a executa operații de I/O către indicatoarele fișier *stdin*, *stdout* și *stderr*. Utilizând aceste fluxuri I/O, puteți executa operații de intrare/ieșire cu șiruri de caractere, valori întregi și valori în virgulă mobilă. Atunci când programele dumneavoastră efectuează operații de I/O utilizând *printf* și *scanf*, funcțiile folosesc specificatorii de format pentru a determina tipurile valorilor (cum ar fi *string*, *int* și așa mai departe). Atunci când efectuați operații de intrare și ieșire cu fluxurile I/O ale limbajului C++, compilatorul furnizează informații despre fiecare tip de valoare, astfel că specificatorii de format nu mai sunt necesari. Un exercițiu interesant poate fi generarea și examinarea unei listări în limbaj de asamblare pentru un fișier, cum ar fi *util_cin.cpp*, care utilizează *cin* și *cout*, în cazul în care compilatorul dumneavoastră acceptă această opțiune.

817 EFECTUAREA IEȘIRILOR CU CLOG

C/C++

După cum ați învățat, C++ dispune de fluxurile *cin*, *cout* și *cerr* care corespund indicatoarelor de fișier *stdin*, *stdout* și *stderr*. În plus, C++ dispune de al patrulea flux I/O denumit *clog*. Fluxul I/O *clog* este similar lui *cerr*, cu deosebirea că acesta efectuează ieșiri prin buffer. Următorul program, *clog.cpp*, utilizează fluxul I/O *clog* pentru a afișa un mesaj:

```
#include <iostream.h>

void main(void)
```

```
{
   clog << "O eroare ciudata de prelucrare";
}
```

FLUXURILE CIN, COUT, CERR ȘI CLOG SUNT INSTANȚE DE CLASĂ

C/C++ 818

Câteva dintre secțiunile precedente au prezentat cum se execută operațiile de I/O cu ajutorul fluxurilor *cin, cout, cerr* și *clog*. Este important să știți că acești identificatori de fluxuri de I/O nu sunt operatori miraculoși construiți în C++. De fapt, *cin, cout, cerr* și *clog* sunt instanțe ale unei *clase I/O*. În secțiunile următoare, veți învăța că o clasă definește un șablon care conține date și metode (funcții sau operații care lucrează cu datele). De aceea, clasa este un mecanism fundamental în C++ pentru programarea pe obiecte. Atunci când începeți crearea propriilor dumneavoastră clase, puteți fi liniștit, știind că ați utilizat mai multe clase până din momentul compilării primului dumneavoastră program. Simbolurile duble „mai mic decât" (<<) și „mai mare decât" (>>) sunt simpli operatori de clasă. Nu fiți îngrijorat dacă acești termeni vă par confuzi, secțiunile următoare îi vor explica în detaliu.

DERULAREA IEȘIRII CU FLUSH

C/C++ 819

Așa cum ați învățat, programele dumneavoastră pot utiliza fluxul I/O *cout* pentru a afișa date către *stdout* și fluxul I/O *clog* pentru a efectua ieșiri prin buffer către *stderr*. Atunci când executați ieșiri prin buffer, ieșirea poate să nu apară pe ecran atât de repede pe cât vă doriți. De obicei, indicatorii de fișier și fluxurile I/O nu trimit ieșirea până nu întâlnesc caracterul *retur de car* sau până nu apare o operație de intrare. În astfel de cazuri, programele dumneavoastră pot utiliza *flush* pentru a trimite imediat ieșirea din buffer. Următorul program, *flush.cpp*, ilustrează modul în care *flush* trimite datele către *stdout* și *stderr*:

```
#include <iostream.h>

void main(void)
  {
    cout << "Acesta va aparea imediat" << flush;
    clog << "\nLa fel si aceasta..." << flush;
  }
```

CONȚINUTUL LUI IOSTREAM.H

C/C++ 820

Toate programele prezentate până acum includ fișierul antet *iostream.h* în locul fișierului *stdio.h*. Așa cum ați învățat în secțiunea 818, fluxurile I/O *cin, cout, cerr* și *clog* sunt de fapt instanțe de clasă. Fișierul *iostream.h* definește clasa de fluxuri corespunzătoare și cei patru identificatori. Nu studiați deocamdată fișierul *iostream.h*. Veți analiza conținutul său mai târziu, când cunoștințele dumneavoastră vă vor permite mai buna lui înțelegere. Deocamdată, însă, trebuie să înțelegeți numai că fișierul *iostream.h* definește biblioteca de clase pentru intrări/ieșiri pentru ecran și tastatură.

821 C++ CERE PROTOTIPURI DE FUNCȚII



În capitolul despre funcții al acestei cărți, ați învățat că *un prototip de funcție* specifică tipul parametrilor pe care o funcție îi primește, precum și tipul valorilor pe care funcția le returnează. Atunci când nu specificați prototipul funcției pentru o funcție în C, compilatorul va genera și va afișa un mesaj de avertizare. În C++, însă, trebuie să specificați prototipul funcției, deoarece, dacă nu-l specificați, programul nu se va compila. Următorul program, *nuproto.cpp*, încearcă să utilizeze funcția *printf* fără să furnizeze tipul funcției (conținut de fișierul antet *stdio.h*):

```
void main(void)
   {
     printf("Acesta nu se compileaza sub C++\n");
   }
```

Dacă încercați să compilați programul *nuproto.cpp*, C++ va genera un mesaj de eroare și compilarea se va încheia.

822 C++ ADAUGĂ NOI CUVINTE CHEIE



După cum ați învățat în secțiunea 31, un *cuvânt cheie* este un identificator care are un înțeles special pentru compilator (astfel de cuvinte cheie sunt *for, while, if* și așa mai departe). În plus față de cuvintele cheie definite de compilatorul de C, tabelul 822 prezintă noile cuvinte cheie pe care le acceptă C++.

Cuvinte cheie C++

asm	*bool*	*catch*	*class*	*delete*	*friend*	*inline*	*mutable*	*namespace*
new	*operator*	*private*	*protected*	*public*	*template*	*this*	*using*	*virtual*

Tabelul 822 *Noile cuvinte cheie acceptate de C++.*

Exact ca în cazul cuvintelor cheie din C, nu puteți utiliza cuvintele cheie în C++ pentru nume de variabile, tipuri sau funcții.

823 C++ ACCEPTĂ UNIUNI ANONIME



După cum ați învățat, o uniune este o structură specială de date pentru care C mapează doi sau mai mulți membri la aceeași locație de memorie. Atunci când declarați o uniune în C, trebuie să declarați o variabilă de tip *uniune*:

```
union Valori
   {
     unsigned datele_mele;
     float datele_lui;
   } solutie;
```

Ulterior, când doriți să stocați date în cadrul uniunii, trebuie să specificați numele variabilei și membrului:

```
solutie.datele_mele = 3;
```

C++, însă, permite programelor dumneavoastră să utilizeze uniuni anonime (sau nedenumite). De exemplu, următorul program, *anonim.cpp*, utilizează o uniune similară celei prezentate mai sus:

```
#include <iostream.h>
void main(void)
   {
     union
       {
         int datele_mele;
         float datele_lui;
       };
     datele_mele = 3;
     cout << "Valoarea lui datele_mele este " << datele_mele;
     datele_lui = 1.2345;
     cout << "\nValoarea lui datele_lui este " << datele_lui;
   }
```

Utilizând uniuni anonime, programele pot elimina supraîncărcările datorate gestionării numelor de uniuni și numelor de membri. Însă, numele membrilor uniunilor anonime trebuie să difere de orice altă variabilă din cadrul domeniului curent de valabilitate. Veți învăța mai multe despre uniunile anonime în secțiunile următoare.

REZOLVAREA DOMENIULUI GLOBAL DE VALABILITATE

C/C++ 824

Așa cum ați învățat, o variabilă globală este recunoscută de la declararea ei până la sfârșitul programului. Atunci când utilizați o variabilă globală, se va întâmpla probabil uneori ca numele variabilei globale să fie același cu un nume de variabilă locală. În astfel de cazuri, funcția va utiliza variabila locală. Uneori, însă, puteți să faceți referință la variabila globală în cadrul funcției care a denumit variabila locală cu un nume similar. Pentru asemenea cazuri, C++ vă permite să precedați numele variabilei globale cu două semne *două puncte*, cum ar fi *::variabila*. Următorul program, *global.cpp*, ilustrează modul de utilizare a *operatorului de rezoluție globală* C++:

```
#include <iostream.h>
int nume_global = 1001;
void main(void)
   {
     int nume_global = 1; // Variabila locala
     cout << "Valoarea variabilei locale " << nume_global << '\n';
     cout << "Valoarea variabilei globale " << ::nume_global << '\n';
   }
```

825 FURNIZAREA VALORILOR IMPLICITE ALE PARAMETRILOR

C/C++

Așa cum ați învățat, parametrii sunt valori transmise către funcții. Principala diferență dintre parametrii funcțiilor din C și C++ este aceea că C++ permite programelor dumneavoastră să obțină valori implicite pentru parametri. Dacă programul va invoca o funcție fără specificarea unuia sau mai multor parametri, programul va utiliza valorile implicite. De exemplu, următorul program, *default.cpp*, utilizează funcția *arata_valori* pentru a afișa trei parametri. Dacă utilizatorul invocă funcția cu mai puțin de trei parametri, programul va utiliza valorile implicite 1, 2 și 3:

```
#include <iostream.h>

void arata_valori(int unu = 1, int doi = 2, int trei = 3)
  {
    cout << unu << ' ' << doi << ' ' << trei << '\n';
  }

void main(void)
  {
    arata_valori(1, 2, 3);
    arata_valori(100, 200);
    arata_valori(1000);
    arata_valori();
  }
```

Observație: *Atunci când omiteți parametrii, nu puteți sări peste un singur parametru. Cu alte cuvinte, atunci când omiteți un parametru, trebuie să omiteți toți parametrii de la dreapta parametrului respectiv.*

826 CONTROLUL DIMENSIUNII IEȘIRII LA COUT

C/C++

Câteva dintre secțiunile acestui capitol au utilizat fluxul I/O *cout* pentru a afișa ieșiri. Atunci când utilizați *cout*, puteți utiliza membrul său *width* pentru a specifica numărul minim de caractere utilizate pentru a afișa ieșirea. De exemplu, următorul program, *setwidth.cpp*, utilizează membrul *width* pentru a selecta dimensiunea minimă a ieșirii la cinci caractere:

```
#include <iostream.h>

void main(void)
  {
    int i;
    for (i = 0; i < 3; i++)
      {
        cout.width(5);
        cout << i << '\n';
      }
  }
```

Atunci când compilaţi şi executaţi programul *setwidth.cpp*, ecranul dumneavoastră va afişa următoarele:

```
0
1
2
C:\>
```

Observaţie: *Când utilizaţi membrul* ***width***, *trebuie să specificaţi valoarea lui* ***width*** *pentru fiecare valoare de la ieşire.*

UTILIZAREA LUI SETW PENTRU FIXAREA DIMENSIUNII LUI COUT

C/C++ 827

În secţiunea 826 aţi utilizat membrul *width* al lui *cout* pentru a specifica numărul minim de caractere utilizate pentru afişarea unei valori. În plus, programele dumneavoastră pot utiliza manipulatorul *setw* pentru a specifica mărimea dorită a textului afişat:

```
#include <iomanip.h>

smanip_int _Cdecl _FARFUNC setw(int dim_dorita);
```

Pentru moment, nu fiţi îngrijorat dacă nu înţelegeţi deplin prototipul funcţiei manipulatorului *setw*. Următorul program, *setw.cpp*, utilizează *setw* pentru a selecta diferite mărimi:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    cout << setw(5) << 1 << '\n' << setw(6) << 2;
    cout << '\n' << setw(7) << 3;
  }
```

Atunci când compilaţi şi executaţi programul *setw.cpp*, ecranul dumneavoastră va afişa următoarele:

```
    1
     2
      3
C:\>
```

Observaţie: *Atunci când utilizaţi manipulatorul* ***setw***, *trebuie să specificaţi dimensiunea dorită pentru fiecare valoare pe care o trimiteţi spre ieşire.*

SPECIFICAREA UNUI CARACTER DE UMPLERE PENTRU COUT

C/C++ 828

În mod implicit, atunci când utilizaţi *cout* cu membrul *width* sau manipulatorul *setw* pentru specificarea umplerii cu caractere, C++ utilizează caracterul spaţiu pentru a umple spaţiile suplimentare. Prin utilizarea membrului *fill* al lui *cout*, programele dumneavoastră pot specifica diferite caractere de umplere. De exemplu, următorul program, *coutpct.cpp*, utilizează punctul drept caracter de umplere:

```
#include <iostream.h>

void main(void)
  {
    int i;
    for (i = 0; i < 3; i++)
      {
        cout.fill('.');
        cout.width(5 + i);
        cout << i << '\n';
      }
  }
```

Atunci când compilaţi şi executaţi programul *coutpct.cpp*, ecranul dumneavoastră va afişa următoarele:

```
....0
.....1
......2
C:\>
```

829 ALINIEREA LA DREAPTA ŞI LA STÂNGA A AFIŞĂRII COUT

C/C++

Aţi învăţat că, folosind manipulatorul *setw* sau membrul *width* al lui *cout*, programele dumneavoastră pot specifica dimensiunea minimă utilizată pentru afişarea unei anumite valori. Când specificaţi o ieşire, programele pot selecta alinierea la dreapta sau la stânga prin utilizarea manipulatorului *setiosflags* şi a membrilor clasei *ios*, ca mai jos:

```
#include <iomanip.h>

smanip_long _Cdecl _FARFUNC setiosflags(long fan);
```

Pentru selectarea alinierii la dreapta cu *setiosflags*, plasaţi următorul manipulator în fluxul *cout*:

```
setiosflags(ios::right)
```

În mod asemănător, pentru selectarea alinierii la stânga, utilizaţi următorul manipulator în fluxul *cout*:

```
setiosflags(ios::left)
```

Pentru moment, nu încercaţi să înţelegeţi formatul acestui cod. În schimb, puteţi utiliza indicatoarele (flags) în programe, ca în următorul exemplu. În secţiunile următoare veţi învăţa cum să utilizaţi clasa *ios*. Următorul program, *dr_st.cpp*, utilizează indicatoarele *ios::right* şi *ios::left* pentru selectarea alinierii la dreapta şi la stânga:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
```

```
{
  int i;
  cout << "Aliniere dreapta\n";
  for (i = 0; i < 3; i++)
    {
      cout.width(5);
      cout << setiosflags(ios::right) <<i;
    }
  cout << "\nAliniere stanga\n";
  for (i = 0; i < 3; i++)
    {
      cout.width(5);
      cout << setiosflags(ios::left) << i;
    }
}
```

După compilarea și executarea programului *dr_st.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aliniere dreapta
    1 2 3
Aliniere stanga
1   2 3
C:\>
```

CONTROLUL NUMĂRULUI DE CIFRE ÎN VIRGULĂ MOBILĂ AFIȘATE DE COUT

C/C++830

Așa cum ați învățat în secțiunea 807, fluxul I/O *cout* permite programelor dumneavoastră să afișeze valori în virgulă mobilă. Atunci când afișați astfel de valori, puteți utiliza manipulatorul *setprecision* pentru specificarea numărului dorit de cifre la dreapta punctului zecimal:

```
#include <iomanip.h>
smanip_int _Cdecl _FARFUNC setprecision(int numar_cifre);
```

Următorul program, *setprec.cpp*, utilizează manipulatorul *setprecision* pentru modificarea numărului de zecimale afișate:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
  {
    int i;
    float valoare = 1.2345;
    for (i = 0; i < 4; i++)
      cout << setprecision(i) << valoare << '\n';
  }
```

După compilarea și afișarea programului *setprec.cpp*, ecranul dumneavoastră va afișa:

```
1.2345
1.2
1.23
1.235
C:\>
```

Așa cum vedeți, dacă specificați precizia 0, *cout* va afișa toate cifrele valorii respective.

831 AFIȘAREA VALORILOR ÎN FORMAT FIX SAU ȘTIINȚIFIC

C/C++

Așa cum ați învățat, fluxul I/O *cout* permite programelor dumneavoastră să afișeze valori în virgulă mobilă. Când utilizați *cout*, puteți selecta formatul fixat sau cel științific (exponențial) pentru a afișa valori în virgulă mobilă. Pentru controlul formatului de afișare a valorilor, programele dumneavoastră pot utiliza indicatoarele *ios::fixed* și *ios::scientific* ale manipulatorului *setiosflags*:

```
#include <iomanip.h>
smanip_long _Cdecl _FARFUNC setiosflags(long fan);
```

Următorul program, *fix.cpp*, utilizează manipulatorul *setiosflags* pentru afișarea formatelor fixat și științific:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
   float valoare = 0.000123;
   cout << setiosflags(ios::fixed) << valoare << '\n';
   cout << setiosflags(ios::scientific) << valoare << '\n';
}
```

După compilarea și afișarea programului *fix.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
0.000123
1.23e-04
C:\>
```

832 RESTABILIREA MODULUI IMPLICIT AL LUI COUT

C/C++

Numeroase secțiuni din acest capitol au utilizat manipulatorul *setiosflags* pentru controlul diverselor opțiuni de formatare ale lui *cout*. Pentru a restabili rapid valorile implicite ale lui *cout*, puteți utiliza manipulatorul *resetiosflags*:

```
#include <iomanip.h>
smanip_long _Cdecl _FARFUNC resetiosflags(long fan);
```

Parametrul *fan* specifică opțiunea pe care vreți să o stabiliți. De exemplu, următorul program, *rstio.cpp*, utilizează manipulatorul *resetiosflags* pentru a reveni de la alinierea la dreapta:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    cout.width(5);
    cout << setiosflags(ios::left) << 5 << '\n';
    cout.width(5);
    cout << 5 << '\n' << resetiosflags(ios::left);
    cout.width(5);
    cout << 1;
  }
```

După compilarea și executarea programului *rstio.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
5
5
    1
C:\>
```

STABILIREA BAZEI PENTRU I/O

C/C++ 833

Ați învățat că utilizând modificatorii *dec*, *oct* și *hex*, puteți selecta valori zecimale, octale și hexazecimale. Dacă programul dumneavoastră trebuie să afișeze diferite valori utilizând o anumită bază, programul poate folosi modificatorul *setbase*. Următorul program, *setbase.cpp*, utilizează modificatorul *setbase* pentru a afișa valoarea 255 utilizând diferite baze:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    cout << setbase(8) << 255 << '\n';
    cout << setbase(10) << 255 << '\n';
    cout << setbase(16) << 255 << '\n';
  }
```

Când compilați și executați programul *setbase.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
377
255
ff
C:\>
```

DECLARAREA VARIABILELOR ACOLO UNDE SUNT NECESARE

C/C++ 834

În limbajul C, programele dumneavoastră pot declara variabile după orice acoladă deschisă. În C++, însă, programele dumneavoastră pot declara variabile la orice locație a programului.

Avantajul unor astfel de declarări este acela că puteți declara variabile mai aproape de locul de utilizare a lor. De exemplu, următorul program, *dec_int.cpp*, declară variabila *int contor* în interiorul buclei *for*, în care programul utilizează variabila:

```
#include <iostream.h>

void main(void)
  {
    cout << "Pe punctul de a incepe bucla\n";
    for (int contor = 0; contor < 10; contor++)
      cout << contor << '\n';
    cout << "valoarea finala a lui contor " << count;
  }
```

Atunci când programul declară o variabilă în cadrul unui bloc, domeniul variabilei începe la punctul declarării ei și se sfârșește la sfârșitul blocului curent și al tuturor blocurilor care apar în cadrul blocului curent.

835 PLASAREA VALORILOR IMPLICITE ALE PARAMETRILOR ÎN PROTOTIPUL FUNCȚIILOR

C/C++

Așa cum ați învățat, C++ vă permite specificarea de valori implicite pentru parametrii funcțiilor. De obicei, astfel de valori implicite apar în antetul funcțiilor:

```
void o_functie(int a = 1, int b = 2, int c = 3)
  {
    cout << a << b << c;
  }
```

În afară de plasarea valorilor implicite ale parametrilor în antetul funcțiilor, programele dumneavoastră pot specifica valorile implicite și în cadrul prototipurilor funcțiilor:

```
void o_functie(int a = 1, int b = 2, int c = 3);
void main(void)
  {
    //instructiuni
  }
```

Dacă funcția pentru care doriți să specificați parametrii impliciți nu se situează în fișierul sursă curent, puteți cere compilatorului să includă valorile implicite corecte prin specificarea valorilor în prototipul funcției, așa cum am arătat mai sus.

836 UTILIZAREA OPERAȚIILOR PE BIȚI ȘI COUT

C/C++

După cum ați învățat, pentru a afișa date la ieșire utilizând fluxul I/O *cout*, programele dumneavoastră utilizează operatori identici cu *operatorul C pe biți de deplasare la stânga* (<<). Următorul program, *biticout.cpp*, utilizează atât operatorul pe biți, cât și *cout*. După cum veți vedea, compilatorul poate decide care operație se efectuează după modul în care programul utilizează operatorul, ca mai jos:

```
#include <iostream.h>
void main(void)
  {
    unsigned int valoare, unu = 1;
    valoare = unu << 1;
    cout << "Valoare: " << valoare << '\n';
    cout << "Rezultat: " << (unu << 1) << '\n';
  }
```

Când compilaţi şi executaţi programul *biticout.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Valoare: 2
Rezultat: 2
C:\>
```

Prin închiderea operaţiei pe biţi între paranteze, compilatorul asociază operatorul cu *operatorul pe biţi de deplasare la stânga*. Dacă aţi elimina parantezele care încadrează operaţia, programul ar produce următorul rezultat:

```
Valoare: 2
Rezultat: 11
C:\>
```

EVALUAREA REDUSĂ

C/C++837

Multe dintre programele prezentate în cadrul acestei cărţi combină condiţii în cadrul construcţiilor *if* şi *while*, ca mai jos:

```
if ((a > 1) && (b < 3))
while ((litera >= 'A') && (litera <= 'Z'))
```

Atunci când programele dumneavoastră combină condiţii în cadrul construcţiilor *if* şi *while*, trebuie să înţelegeţi că C++ generează cod care efectuează o evaluare *redusă*. Evaluarea redusă înseamnă că, în cazul în care rezultatul uneia dintre condiţii va face întreaga condiţie adevărată sau falsă, programul va opri evaluarea celorlalte condiţii. De exemplu, fiind dată precedenta instrucţiune *if*, programul nu va compara variabila *b* cu valoarea 3, dacă prima parte a condiţiei *if* eşuează. Efectuarea celei de a doua comparaţii ar fi consumat inutil timpul procesorului. În mod asemănător, fiind dată precedenta instrucţiune *while*, programul nu va compara variabila *litera* cu 'Z', dacă litera nu este mai mare sau egală cu 'A'. Efectuarea evaluării reduse permite programelor să economisească din timpul de procesare. Totuşi, dacă nu sunteţi conştient de faptul că programele dumneavoastră efectuează o astfel de prelucrare, pot apărea erori. De exemplu, consideraţi următoarea instrucţiune *if*:

```
if ((valoare < 10) && ((litera = getchar()) != 'Q'))
```

În acest caz, dacă variabila valoare nu este mai mică decât 10, programul nu va efectua cealaltă comparaţie care utilizează funcţia macro *getchar* pentru a atribui o valoare variabilei *litera*. Ca rezultat, uneori instrucţiunea *if* va atribui o valoare variabilei *litera*, iar alteori nu.

838 UTILIZAREA CUVÂNTULUI CHEIE CONST ÎN C++

După cum ați învățat, cuvântul cheie *const* informează compilatorul de faptul că programul nu trebuie să modifice variabila ce urmează, în cursul execuției programului. Atunci când utilizați cuvântul cheie *const* în programele C++, puteți utiliza variabila corespondentă în orice modalitate în care ați fi putut utiliza în mod normal o expresie constantă. De exemplu, următoarele instrucțiuni utilizează constanta *dim_sir* pentru a specifica dimensiunea unei matrice de șiruri de caractere:

```
const int dim_sir = 64;
char sir[dim_sir];
```

Avantajul utilizării constantelor față de macrodefinițiile create cu *#define* este că aceste constante vă permit să specificați tipul informației.

839 UTILIZAREA CUVÂNTULUI CHEIE ENUM ÎN C++

După cum ați învățat, cuvântul cheie *enum* permite programelor dumneavoastră să definească tipuri enumerate. Cuvântul cheie *enum* este în C++ foarte asemănător celui utilizat de C, exceptând faptul că atunci când declarați un tip enumerat în C++, programul dumneavoastră va putea, mai târziu, să utilizeze eticheta tipului ca un tip. De exemplu, următoarea instrucțiune declară în limbajul C o variabilă denumită *zi*:

```
enum Zile { luni, marti, miercuri, joi, vineri };
enum Zile zi;
```

În C++ declarația va deveni:

```
enum Zile { luni, marti, miercuri, joi, vineri };
Zile zi;
```

După cum puteți vedea, cea de a doua declarație nu impune utilizarea cuvântului cheie *enum* înaintea numelui de tip *Zile*.

840 SPAȚIUL LIBER



După cum ați învățat în capitolul despre memorie al acestei cărți, programele dumneavoastră pot să aloce în mod dinamic memorie heap în timpul execuției. Citind documentația limbajului C++, veți întâlni termenul de spațiu liber. Heap și spațiu liber sunt același lucru. Pentru a aloca memorie din spațiul liber, programele C++ utilizează *new* și *delete*. Este important să observați că, spre deosebire de *malloc* și *free*, care sunt funcții, *new* și *delete* sunt operatori. Secțiunea 841 prezintă modul de utilizare a operatorului *new* pentru alocarea memoriei.

841 ALOCAREA MEMORIEI CU NEW



După cum ați învățat, programele C++ alocă memorie dinamică din spațiul liber cu ajutorul operatorului *new*. Pentru a utiliza *new*, programul trebuie să specifice numărul dorit de octeți. Următorul program, *new_sir.cpp*, alocă memorie pentru o matrice de 256 de octeți. Apoi programul completează matricea cu litera A și îi afișează conținutul:

```
#include <iostream.h>
void main(void)
 {
   char *matrice = new char[256];
   int i;
   for (i = 0; i < 256; i++)
     matrice[i] = 'A';
   for (i = 0; i < 256; i++)
     cout << matrice[i] << ' ';
 }
```

ALOCAREA MAI MULTOR MATRICE

C/C++842

În secțiunea 841 ați utilizat operatorul *new* pentru a aloca dinamic o matrice de șiruri de caractere de 256 de octeți. După aceasta, compilatorul alocă memorie când programul declară variabila pointer:

```
char *matrice = new char[256];
```

Programele dumneavoastră pot utiliza operatorul *new* pentru a aloca memorie, din orice locație. Următorul program, *new_cop.cpp*, utilizează operatorul *new* pentru a aloca trei șiruri de caractere, fiecare la o locație diferită în cadrul programului:

```
#include <iostream.h>
void main(void)
 {
   char *matrice = new char[256];
   char *tinta, *destinatie;
   int i;
   tinta = new char[256];
   for (i = 0; i < 256; i++)
     {
       matrice[i] = 'A';
       tinta[i] = 'B';
     }
   destinatie = new char[256];
   for (i = 0; i < 256; i++)
     {
       destinatie[i] = tinta[i];
       cout << destinatie[i] << ' ';
     }
 }
```

843 TESTAREA EXISTENȚEI SPAȚIULUI LIBER

C/C++

În capitolul despre memorie al acestei cărți ați învățat că atunci când memoria heap nu poate satisface o cerere, funcțiile *calloc și malloc* returnează *NULL*. Același lucru se întâmplă și în cazul operatorului *new* și al spațiului liber. Următorul program, *neliber.cpp*, utilizează operatorul *new* pentru a aloca memorie până când se consumă tot spațiul liber:

```
#include <iostream.h>

void main(void)
  {
    char *pointer;
    do
     {
       pointer = new char[10000];
       if (pointer)
         cout << "10000 octeti alocati\n";
       else
         cout << "Alocare esuata\n";
     } while (pointer);
  }
```

844 CONSIDERAȚII DESPRE SPAȚIUL DIN MEMORIA HEAP

C/C++

Așa cum ați învățat, C++ se referă la heap ca spațiu liber. În funcție de modelul de memorie utilizat, cantitatea disponibilă de spațiu heap va diferi. De exemplu, puteți compila programul *neliber.cpp* utilizând modelul de memorie *large*. În acest caz, programul *neliber.cpp*, compilat cu modelul *large* de memorie, se va executa într-un timp mai îndelungat și va aloca cu mult mai mult spațiu de heap înainte să eșueze.

845 UTILIZAREA POINTERILOR FAR ȘI A OPERATORULUI NEW

C/C++

După cum ați învățat, operatorul *new* permite programelor dumneavoastră să aloce memorie din spațiul liber. Dacă utilizați modelul *small* de memorie, spațiul liber corespunde memoriei heap apropiată (near). Dacă programele dumneavoastră trebuie să aloce mai multă memorie decât permite spațiul heap apropiat, programele dumneavoastră pot aloca pointeri far. Următorul program, *new_far.cpp*, alocă pointeri far din spațiul liber, până când spațiul heap îndepărtat (far) rămâne fără memorie:

```
#include <iostream.h>

void main(void)
  {
    char far *pointer;
    do
     {
```

```
    pointer = new far char[10000];
    if (pointer)
      cout << "10000 octeti alocati\n";
    else
      cout << "Alocare esuata\n";
   } while (pointer);
}
```

ELIBERAREA MEMORIEI PENTRU SPAȚIUL LIBER

C/C++ 846

După cum ați învățat, atunci când programele dumneavoastră alocă memorie dinamic, ele ar trebui să elibereze memoria de îndată ce nu mai au nevoie de ea. Când programele dumneavoastră în C alocă memorie utilizând *calloc* și *malloc*, ele eliberează memoria prin *free*. Când programele dumneavoastră în C++ alocă memorie utilizând *new*, ele ar trebui să elibereze mai târziu memoria utilizând *delete*. Următorul program, *delete.cpp*, utilizează operatorul *delete* pentru a elibera trei matrice alocate dinamic pentru spațiul liber:

```
#include <iostream.h>

void main(void)
  {
    char *matrice = new char[256];
    char *tinta, *destinatie;
    int i;
    tinta = new char[256];
    for (i = 0; i < 256; i++)
      {
        matrice[i] = 'A';
        tinta[i] = 'B';
      }
    delete matrice;
    destinatie = new char[256];
    for (i = 0; i < 256; i++)
      {
        destinatie[i] = tinta[i];
        cout << destinatie[i] << ' ';
      }
    delete tinta;
    delete destinatie;
  }
```

REFERINȚELE ÎN C++

C/C++ 847

Un *alias* este un alt nume dat unei variabile. În C, programele dumneavoastră pot crea un alias utilizând pointerii. C++ simplifică crearea de aliasuri utilizând referințele. Pentru a crea o referință, utilizați operatorul de referențiere (&):

```
int variabila;
int& alias = variabila;
```

Operatorul de referențiere este similar cu operatorul C de adresare. Observați, însă, poziționarea operatorului. Operatorul de referențiere urmează imediat unui tip (cum sunt *int*, *float* sau *char*). Fiecare alias poate corespunde numai unei singure variabile de-a lungul existenței sale. Următorul program, *alias.cpp*, creează două aliasuri și le utilizează pentru a afișa adresele variabilelor specificate:

```
#include <iostream.h>

void main(void)
  {
    int a = 1001;
    int& a_alias = a;
    float pret = 39.95;
    float& pret_alias = pret;
    cout << "Valoarea lui a e " << a << " aliasul e " << a_alias;
    cout << "\nPretul e " << pret << " aliasul e " << pret_alias;
    a_alias++;
    cout << "\nValoarea lui a e " << a << " aliasul e " << a_alias;
  }
```

Programul *alias.cpp* utilizează variabila de referențiere *a_alias* pentru a incrementa valoarea lui *a*. Atunci când un program se referă la o referință, orice operație va corespunde direct variabilei alias.

848 TRANSMITEREA UNEI REFERINȚE CĂTRE O FUNCȚIE

C/C++

După cum ați învățat, pentru a modifica o variabilă în cadrul unei funcții, programele dumneavoastră trebuie să transmită un pointer către variabilă. Când utilizați C++, puteți să simplificați modificarea unei variabile în cadrul unei funcții, folosind o *referință*. Utilizând o referință, eliminați necesitatea operatorului pointer (->). Următorul program, *functref.cpp*, transmite o referință la variabila *valoare* către funcția *modif_val*, care atribuie variabilei valoarea 1500:

```
#include <iostream.h>

void modif_val(int& val_referinta)
  {
    val_referinta = 1500;
  }
void main(void)
  {
    int valoare = 10;
    int& alias = valoare;
    cout << "Valoarea inaintea functiei: " << valoare << '\n';
    modif_val(alias);
    cout << "Valoarea dupa functie: " << valoare <<'\n';
  }
```

După cum vedeți, utilizând o referință, simplificați procesul de modificare a unei valori în cadrul unei funcții.

ATENȚIE LA OBIECTELE ASCUNSE

C/C++ 849

După cum ați învățat, o referință creează un alt nume pentru o variabilă – un alias. Atunci când creați referința, trebuie să vă asigurați că tipul referinței este identic cu tipul la care se face referirea. De exemplu, următoarea instrucțiune creează o referință la o variabilă de tip *int*:

```
int valoare;
int& alias = valoare;
```

Dacă tipul referinței și tipul variabilei diferă, C++ va crea un *obiect ascuns* care nu este alias la valoarea specificată, ci stochează în schimb valoarea pentru o variabilă nedenumită de tipul referinței. De exemplu, următoarea instrucțiune creează un obiect ascuns de tip *float*:

```
int valoare;
float& alias = valoare;
```

După cum observați, tipurile referinței și variabilei diferă. De aceea, compilatorul nu va crea un alias la variabila *valoare*, ci va aloca memorie pentru o valoare în virgulă mobilă, creând în memorie un alias pentru referința dată. Motivul pentru care e nevoie de atenție la obiectele ascunse este acela că acestea pot conduce la erori care sunt foarte dificil de detectat. Dacă modificați tipul variabilei, asigurați-vă de asemenea de modificarea tipului referinței corespunzătoare, dacă aceasta există.

UTILIZAREA A TREI MODALITĂȚI DE TRANSMITERE A PARAMETRILOR

C/C++ 850

În C, programele dumneavoastră pot transmite parametri către funcții utilizând *apelarea prin valoare* și *apelarea prin pointer de referință*. După cum ați învățat, în C++ programele dumneavoastră pot utiliza o a treia tehnică, *apelarea prin referință*. Următorul program, *apel_3.cpp*, ilustrează modul în care se utilizează toate cele trei tehnici:

```
#include <iostream.h>
#include <iomanip.h>

void apel_prin_valoare(int a, int b, int c)
  {
    a = 3; b = 2; c = 1;
  }
void apel_prin_pointer_referinta(int *a, int *b, int *c)
  {
    *a = 3; *b = 2; *c = 1;
  }
void apel_prin_referinta(int& a, int& b, int& c)
  {
    a = 1; b = 2; c = 3;
  }
```

```
void main(void)
  {
    int a = 1, b = 2, c = 3;
    int& a_alias = a;
    int& b_alias = b;
    int& c_alias = c;
    apel_prin_valoare(a, b, c);
    cout << "Prin valoare: " << a << b << c < '\n';
    apel_prin_pointer_referinta(&a, &b, &c);
    cout << "Prin pointer: " << a << b << c << '\n';
    apel_prin_referinta(a_alias, b_alias, c_alias);
    cout << "Prin referinta: " << a << b << c < '\n';
  }
```

851 REGULI PENTRU LUCRUL CU REFERINȚELE

C/C++

În C++, o referință permite crearea unui alias pentru o variabilă. Atunci când utilizați referințe, rețineți următoarele reguli:

1. După inițializare, programul dumneavoastră nu poate modifica valoarea referință.
2. Tipul referinței și tipul variabilei trebuie să fie aceleași.
3. Nu puteți crea un pointer la o referință.
4. Nu puteți compara valoarea a două referințe – comparațiile ar compara valorile variabilelor referențiate.
5. Nu puteți incrementa, decrementa sau modifica valoarea referință – operațiile se vor aplica valorii variabilelor referențiate.
6. Puteți distinge operatorul de referențiere de operatorul de adresare, deoarece operatorul de referențiere urmează întotdeauna tipului (de exemplu, *int&*).

852 FUNCȚIILE POT RETURNA REFERINȚE

C/C++

În C++, o referință este un alias pentru o variabilă. După cum ați învățat, referințele pot simplifica transmiterea de parametri prin eliminarea necesității efectuării operațiilor cu pointeri. Toate secțiunile prezentate până acum au inițializat variabile referință la declarare, chiar la începutul blocului *main*. C++ permite însă funcțiilor să returneze referințe. Deoarece C++ permite programelor dumneavoastră să declare variabile la orice locație, programele pot crea și inițializa o referință la orice locație din program prin returnarea unei referințe de la o funcție. Următorul program, *rtn_ref.cpp*, invocă funcția *da_carte*, care returnează o referință la o variabilă de tip *carte*:

```
#include <iostream.h>

struct carte
  {
    char autor[64];
    char titlu[64];
```

```
      float pret;
   };
carte biblioteca[3] = {
      {"Jamsa si Klander", "Totul despre C/C++", 49.95},
      {"Klander", "Hacker Proof", 54.95},
      {"Jamsa si Klander", "Visual Basic", 54.95}};
carte& da_carte(int i)
   {
      if ((i >= 0) && (i < 3))
        return(biblioteca[i]);
      else
        return(biblioteca[0]);
   }
void main(void)
   {
      cout << "Pe punctul de a obtine cartea 0\n";
      carte& o_carte = da_carte(0);
      cout << o_carte.autor << ' ' << o_carte.titlu;
      cout << ' ' << o_carte.pret;
   }
```

UTILIZAREA CUVÂNTULUI CHEIE INLINE

C/C++ 853

După cum ați învățat, programele transmit parametri către funcții utilizând stiva. De fiecare dată când programele dumneavoastră apelează o funcție, calculatorul trebuie să depună parametrii funcției (și adresa de returnare a programului) în stivă și apoi să extragă aceleași valori din stivă. Aceste operații de depunere și extragere conduc la o supraîncărcare care face utilizarea funcțiilor puțin mai lentă decât utilizarea codului *inline*. Dacă programele dumneavoastră conțin una sau două funcții importante care trebuie să se execute rapid, ar trebui să utilizați cuvântul cheie *inline* pentru a cere compilatorului să plaseze codul corespunzător *inline*, la fiecare apelare a funcției și nu să creați coduri distincte de funcție. Dacă programul dumneavoastră apelează funcția *inline* din cinci locații diferite, compilatorul va insera funcția corespunzătoare în program de cinci ori. Dacă programul dumneavoastră apelează funcția *inline* din 50 de locații diferite, compilatorul inserează codul de 50 de ori. De aceea, codul *inline* afectează performanțele de timp și spațiu. Utilizând cod *inline*, creați programe mai rapide, dar faceți codul programului să fie mai mare (ceea ce, teoretic, poate încetini programul). Următorul program, *inline.cpp*, utilizează două funcții similare, plasând una *inline* și apelând-o pe cealaltă. Programul afișează durata necesară pentru apelarea de 30000 de ori a fiecărei funcții:

```
#include <iostream.h>
#include <time.h>

inline void interschimb_inline(int *a, int *b, int *c, int *d)
   {
      int temp;
      temp = *a;
      *a = *b;
```

```
      *b = temp;
      temp = *c;
      *c = *d;
      *d = temp;
   }
void apel_interschimb(int *a, int *b, int *c, int *d)
   {
      int temp;
      temp = *a;
      *a = *b;
      *b = temp;
      temp = *c;
      *c = *d;
      *d = temp;
   }
void main(void)
   {
      clock_t start, stop;
      long int i;
      int a = 1, b = 2, c = 3, d = 4;
      start = clock();
      for (i = 0; i < 300000L; i++)
        interschimb_inline(&a, &b, &c, &d);
      stop = clock();
      cout << "Durata pentru inline: " << stop - start;
      start = clock();
      for (i = 0; i < 300000L; i++)
        apel_interschimb(&a, &b, &c, &d);
      stop = clock();
      cout << "\nDurata pentru functia apelata: " << stop - start;
   }
```

854 UTILIZAREA CUVÂNTULUI CHEIE ASM

C/C++

După cum ați învățat, în funcție de scopurile programului dumneavoastră, probabil că uneori va trebui să programați la nivel inferior, în limbaj de asamblare. În aceste cazuri, puteți crea o funcție în limbaj de asamblare și să legați funcția la program sau puteți să utilizați cuvântul cheie *asm* pentru a insera instrucțiuni în limbaj de asamblare în codul dumneavoastră în C++. Următorul program, *asm_demo.cpp*, utilizează cuvântul cheie *asm* pentru a include instrucțiuni în limbaj de asamblare necesare pentru a face difuzorul încorporat al calculatorului să emită sunete:

```
#include <iostream.h>
void main(void)
   {
      cout << "Pe punctul de a emite un sunet...\n";
```

```
   asm
     {
       MOV AX,0x0200
       MOV DL,7
       INT 0x21
     };
   cout << "Gata..\n";
 }
```

CITIREA UNUI CARACTER UTILIZÂND CIN

C/C++855

Câteva dintre secțiunile precedente au utilizat fluxul I/O *cin* pentru a citi intrarea de la tastatură. Pentru a mări controlul asupra introducerii de la tastatură sau a introducerii redirectate, programele dumneavoastră pot utiliza *cin.get* pentru a citi caracterele unul câte unul:

```
caracter = cin.get();
```

Următorul program, *cin_get.cpp*, utilizează *cin.get* pentru a atribui caracterele până la, dar nu inclusiv, caracterul de linie nouă șirului de caractere *str*:

```
#include <iostream.h>
#include <stdio.h>

void main(void)
  {
    char sir[256];
    int i = 0;
    while ((sir[i] = cin.get()) != '\n')
      i++;
    sir[i] = NULL;
    cout << "Sirul a fost: " << sir;
  }
```

SCRIEREA UNUI CARACTER CU COUT

C/C++856

În secțiunea 855 ați învățat că programele dumneavoastră pot primi la intrare caractere unul câte unul utilizând *cin.get*. În mod similar, programele dumneavoastră pot utiliza *cout.put* pentru a scrie un caracter:

```
cout.put(caracter);
```

Următorul program, *cout_put.cpp*, utilizează *cout.put* pentru a afișa caracterele dintr-un șir, unul câte unul:

```
#include <iostream.h>
void main(void)
```

```
{
  char *titlu = "Totul despre C/C++";
  while (*titlu)
    cout.put(*titlu++);
}
```

857 SCRIEREA UNUI PROGRAM FILTRU SIMPLU

C/C++

După cum ați învățat, *cout.put* și *cin.get* permit programelor dumneavoastră să efectueze operații de I/O cu caractere. Următorul program, *lit_maj.cpp*, convertește intrarea redirectată în majuscule. Pentru a efectua conversia, programul ciclează pur și simplu până când *cin.get* returnează –1, indicând sfârșitul fișierului:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
  char litera;
  while ((litera = cin.get()) != -1)
    cout.put(toupper(litera));
}
```

858 SCRIEREA UNEI COMENZI SIMPLE TEE

C/C++

După cum ați învățat, C++ vă permite redirectarea ieșirii fluxului I/O *cout*. Următorul program, *tee.cpp*, scrie intrările sale redirectate la fluxurile I/O *cout* și *cerr*. Pentru că programul utilizează două fluxuri I/O, puteți vedea intrarea programului pe ecran și puteți totuși redirecta ieșirea către altă sursă:

```
#include <iostream.h>

void main(void)
{
  char litera;
  while ((litera = cin.get()) != -1)
    {
      cout.put(litera);
      cerr.put(litera);
    }
}
```

859 SCRIEREA UNEI COMENZI SIMPLE FIRST

C/C++

După cum ați învățat, fluxurile I/O *cout* și *cin* acceptă redirectarea I/O. Următorul program, *first1.cpp*, utilizează aceste fluxuri de intrare pentru a scrie primele zece linii ale intrării redirectate pe ecran:

```
#include <iostream.h>
void main(void)
  {
    char litera;
    int contor = 0;
    while ((litera = cin.get()) != -1)
      {
        cout.put(litera);
        if ((litera == '\n') && (++contor == 10))
          break;
      }
  }
```

SCRIEREA UNEI COMENZI FIRST PERFECȚIONATE

C/C++ 860

În secțiunea 859 ați creat programul *first1.cpp* care afișa primele zece linii ale intrării redirectate. O comandă mai flexibilă ar permite utilizatorului să specifice ca argument al liniei de comandă, numărul de linii pe care utilizatorul dorește să le afișeze. Următorul program, *first2.cpp*, permite utilizatorului să realizeze acest lucru:

```
#include <iostream.h>
#include <stdlib.h>

void main(int argc, char **argv)
  {
    char litera;
    int contor = 0;
    int limita_linie;
    limita_linie = atoi(argv[1]);
    while ((litera = cin.get()) != -1)
      {
        cout.put(litera);
        if ((litera == '\n') && (++contor == limita_linie))
          break;
      }
  }
```

Dacă utilizatorul nu specifică numărul de linii pe care să le afișeze programul sau dacă specifică un număr de linii incorect, programul va afișa toată intrarea redirectată.

TESTAREA SFÂRȘITULUI DE FIȘIER

C/C++ 861

Câteva dintre secțiunile precedente au utilizat *cin.get* pentru a determina capătul intrării redirectate, ca mai jos:

```
while ((litera = cin.get()) != -1)
```

În plus față de testarea valorii returnate de metoda *cin.get*, programele dumneavoastră pot testa cu *cin.eof*, ca mai jos:

```
while (! cin.eof())
```

Următorul program, *firsteof.cpp*, modifică programul *firstz.cpp* pentru a testa capătul de fișier cu metoda *cin.eof*.

```
#include <iostream.h>
#include <stdlib.h>

void main(int argc, char **argv)
  {
    char litera;
    int contor = 0;
    int limita_linie;
    limita_linie = atoi(argv[1]);
    while (! cin.eof())
      {
        litera = cin.get();
        cout.put(litera);
        if ((litera == '\n') && (++contor == limita_linie))
          break;
      }
  }
```

862 GENERAREA UNEI LINII NOI CU ENDL

C/C++

Multe dintre secțiunile anterioare au plasat caracterul de linie nouă (\n) în fluxul de ieșire *cout* pentru a genera *retur de car* și *avans de rând*. În plus față de utilizarea caracterului de linie nouă, programele dumneavoastră pot utiliza *endl*, ca mai jos:

```
cout << "Salutari!" << endl;
```

Următorul program, *endl.cpp*, utilizează *endl* de câteva ori pentru a genera *retur de car* și *avans de rând*:

```
#include <iostream.h>

void main(void)
  {
    cout << "Aceasta este linia unu" << endl;
    cout << "Aceasta este linia doi" << endl;
    cout << "Aceasta este linia trei--";
    cout << "Este ultima linie" << endl;
  }
```

Când compilați și executați programul *endl.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este linia unu
Aceasta este linia doi
Aceasta este linia trei--Este ultima linie
C:\>
```

SPECIFICĂRILE PENTRU EDITAREA LEGĂTURILOR

C/C++863

După cum ați învățat, C++ cere prototipuri de funcții pentru fiecare funcție utilizată de programele dumneavoastră. Compilatorul de C++ utilizează prototipurile pentru a verifica tipurile parametrilor și ale valorilor returnate. În timpul compilării, compilatorul de C++ modifică numele funcțiilor și parametrii lor în codul obiect care rezultă. Editorul de legături, la rândul său, utilizează aceste noi nume pentru a rezolva referințele externe. Din păcate, dacă editați legături cu un cod care a fost anterior compilat de un compilator de C, numele funcțiilor în codul obiect nu vor fi de același „format de nume de funcție C++". Pentru a preveni compilatorul de C++ să nu modifice numele funcțiilor C, puteți utiliza *specificatorul de legături*. Pe scurt, specificatorul de legături spune compilatorului C++ formatul corect pe care ar trebui să-l utilizeze pentru denumirea funcțiilor în fișierul obiect. Să presupunem de exemplu, că aveți o funcție denumită *calcul_plati* pe care dumneavoastră (sau alt programator) ați scris-o anterior în C. Pentru a cere compilatorului de C++ să nu modifice formatul de nume al funcției, trebuie să utilizați următorul specificator de legaturi:

```
extern "C"
  {
    float calcul_plati(int nr_angaj, char *fis_angaj);
  };
```

Observație: *Dacă examinați fișierele antet pe care vi le pune la dispoziție compilatorul, veți găsi câțiva specificatori de legături în fișiere care sunt similari cu cel prezentat în această secțiune.*

SUPRAÎNCĂRCAREA

C/C++864

Supraîncărcarea este procesul de atribuire a mai mult de o operație unui operator sau utilizarea a două sau mai multe funcții cu același nume. De exemplu, C și C++ utilizează simbolul plus (+) ca operator de *adunare*. Puteți utiliza supraîncărcarea pentru a cere ca C++ să folosească simbolul plus pentru șiruri concatenate:

```
numecale = nume_director + numefisier;
```

În funcție de modalitatea în care programul dumneavoastră folosește simbolul plus, compilatorul de C++ va determina dacă instrucțiunea de program efectuează adunarea sau concatenarea șirurilor. Ați învățat, de asemenea, că atunci când utilizați C, uneori trebuie să creați funcții denumite diferit pentru a lucra cu valori de diferite tipuri. De exemplu, dacă ați creat o funcție care returnează suma valorilor unei matrice de valori întregi, trebuie să creați o funcție cu un nume diferit dacă doriți să însumați valorile dintr-o matrice de tip *float*. După cum ați învățat, C++ vă permite să supraîncărcați funcțiile și operatorii, ceea ce simplifică multe operații.

865 SUPRAÎNCĂRCAREA FUNCȚIILOR

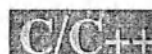


După cum ați învățat, C++ vă permite să aveți mai multe funcții cu același nume. În timpul compilării, compilatorul de C++ determină care funcție trebuie apelată, bazându-se pe numărul și tipul parametrilor pe care instrucțiunea de apelare îi transmite funcției. De exemplu, următorul program, *over.cpp*, creează două funcții numite *suma* care returnează suma numerelor de elemente dintr-o matrice. Prima funcție acceptă matrice de tip *float*, în timp ce cea de a doua acceptă matrice de tip *int*:

```
#include <iostream.h>

int suma(int *matrice, int nr_element)
   {
     int rezultat = 0;
     int nr;
     for (nr = 0; nr < nr_element; nr++)
       rezultat += matrice[nr];
     return(rezultat);
   }
float suma(float *matrice, int nr_element)
   {
     float rezultat = 0;
     int nr;
     for (nr = 0; nr < nr_element; nr++)
       rezultat += matrice[nr];
     return(rezultat);
   }
void main(void)
   {
     int a[5] = { 1, 2, 3, 4, 5 };
     float b[4] = { 1.11, 2.22, 3.33, 4.44 };
     cout << "Suma valorilor int: " << suma(a, 5) << '\n';
     cout << "Suma valorilor float: " << suma(b, 4) << '\n';
   }
```

866 SUPRAÎNCĂRCAREA FUNCȚIILOR: UN AL DOILEA EXEMPLU



După cum ați învățat, C++ vă permite supraîncărcarea funcțiilor, prin crearea în programele dumneavoastră a două sau mai multe funcții care au același nume. Următorul program supraîncarcă funcția *interschimb*. Prima funcție interschimbă două valori, în timp ce a doua interschimbă patru. În timpul compilării, compilatorul utilizează numărul de parametri pentru a determina ce funcție apelează. Programul *utparam.cpp* este primul dumneavoastră program de supraîncărcare a funcțiilor:

```
#include <iostream.h>

void interschimb(int *a, int *b)
```

```
  {
  int temp = *a;
  *a = *b;
  *b = temp;
  }
void interschimb(int *a, int *b, int *c, int *d)
  {
  int temp = *a;
  *a = *b;
  *b = temp;
  temp = *c;
  *c = *d;
  *d = temp;
  }
void main(void)
  {
  int a = 1, b = 2, c = 3, d = 4;
  interschimb(&a, &b);
  cout << "A interschimbat a cu b " << a << b << '\n';
  interschimb(&a, &b, &c, &d);
  cout << "A interschimbat patru " << a << b << c << d << '\n';
  }
```

EVITAREA AMBIGUITĂȚII LA SUPRAÎNCĂRCARE

C/C++ 867

Atunci când creați funcții supraîncărcate, este posibil să apară o situație în care compilatorul este incapabil să distingă între două (sau mai multe) funcții supraîncărcate. Atunci când creați două sau mai multe funcții supraîncărcate între care compilatorul este incapabil să facă distincția, compilatorul consideră funcțiile *ambigue*. Apelările funcțiilor ambigue sunt erori, iar compilatorul nu va compila programul.

De departe cea mai întâlnită cauză de ambiguitate este conversia automată a tipurilor în C++. După cum ați învățat, C++ încearcă în mod automat să convertească argumentele utilizate de program pentru apelarea funcției în tipul de argumente pe care funcția le așteaptă. De exemplu, să vedem următorul fragment de cod:

```
int functmea(double d);
//
//Codul program
//
cout << functmea('c'); // C++ converteste in intregi
```

Așa cum arată comentariul din fragmentul de cod, apelul funcției din exemplul dat nu cauzează o eroare datorită conversiei automate în C++ a caracterului la echivalentul său *double*. Puține conversii de tip de felul celei prezentate în exemplul precedent nu sunt admise în C++. În timp ce conversiile sunt convenabile, ele cauzează serioase probleme în cazul funcțiilor supraîncărcate. Următorul program, *over_er.cpp*, supraîncarcă funcția

ex_functie cu două tipuri diferite de parametri, *float* și *double*. Programul apelează funcția de două ori – o dată cu valoarea 1500.1, care este un parametru de tip *double* și de aceea nu provoacă ambiguitate; iar altă dată cu valoarea 1500, care provoacă ambiguitate deoarece compilatorul nu știe dacă ar trebui să convertească valoarea în *float* sau în *double*. Atunci când compilați următorul program, veți primi un mesaj de eroare la compilare:

```
#include <iostream.h>

float ex_functie(float i);
double ex_functie(double i);
void main(void)
  {
    cout << ex_functie(1500.1) << " ";    // neambigua, apeleaza
                                          // ex_functie(double)
    cout << ex_functie(1500);             // ambigua
  }
float ex_functie(float i)
  {
    return i;
  }
double ex_functie(double i)
  {
    return -i;
  }
```

Observație: *În secțiunea 814 ați învățat să nu transmiteți o referință pentru* ***cin***. *După cum ați învățat în această secțiune, transmiterea unei referințe către o funcție supraîncărcată va produce încurcături compilatorului. Deoarece poate procesa mai multe tipuri de valori,* ***cin*** *trebuie, de asemenea, să fie o funcție supraîncărcată – ceea ce explică motivul pentru care nu se pot folosi apelările prin referință.*

868 CITIREA LINIE CU LINIE UTILIZÂND CIN

C/C++

După cum ați învățat, programele dumneavoastră pot citi intrările de la tastatură utilizând *cin*. Atunci când programele dumneavoastră doresc să citească intrările caracter cu caracter, ele pot folosi *cin.get*. În anumite cazuri, programele dumneavoastră pot necesita efectuarea operațiilor de intrare linie cu linie. Pentru asemenea cazuri, programele dumneavoastră pot utiliza *cin.getline*:

```
char sir[256];
cin.getline(sir, sizeof(sir), '\n');
```

Parametrul *sir* este un pointer la șirul de caractere pe care doriți să-l citească *cin.getline*. Operatorul *sizeof* specifică numărul de octeți pe care șirul îi poate stoca. În sfârșit, caracterul de linie nouă arată caracterul care va încheia citirea. Următorul program, *getline.cpp*, utilizează *cin.getline* pentru a citi o linie de intrare:

```
#include <iostream.h>

void main(void)
```

```
{
  char sir[256];
  cout << "Introduce numele si prenumele si apasa Enter\n";
  cin.getline(sir, sizeof(sir), '\n');
  cout << sir;
}
```

Pentru a testa funcția *cin.getline*, modificați caracterul de încheiere cu o literă din alfabet (spre exemplu) și apoi observați rezultatul programului.

UTILIZAREA LUI CIN.GETLINE ÎNTR-O BUCLĂ

C/C++ 869

În secțiunea 857, programul dumneavoastră a folosit *cin.get* și *cout.put* pentru a afișa o intrare redirectată cu majuscule. Următorul program, *lit_maj2.cpp*, utilizează *cin.getline* și *cout* pentru a executa un proces similar:

```
#include <iostream.h>
#include <string.h>

void main(void)
{
  char sir[256];
  while (cin.getline(sir, sizeof(sir), '\n'))
    cout << strupr(sir) << '\n';
}
```

După cum vedeți, programul ciclează până *cin.getline* returnează 0, ceea ce indică sfârșitul intrării redirectate. Deoarece funcția *cin.getline* nu plasează caracterul de linie nouă în șirul de caractere, *cout* trebuie să scrie caracterul de linie nouă pentru fiecare linie.

MODIFICAREA CONTROLULUI IMPLICIT AL OPERATORULUI NEW

C/C++ 870

După cum ați învățat, atunci când operatorul *new* nu poate atribui suficientă memorie pentru a satisface solicitările de memorie, *new* returnează *NULL*. În raport de funcția programului dumneavoastră, e posibil ca *new* să efectueze alte prelucrări atunci când nu poate aloca memorie. Așa cum reiese, atunci când *new* nu poate aloca memorie, el poate să apeleze funcția indicată de un pointer global la o funcție denumită *_new_handler*. Prin atribuirea variabilei *_new_handler* să indice o funcție personalizată, puteți indica operatorului *new* să apeleze funcția dumneavoastră proprie atunci când *new* nu poate aloca memorie. Următorul program, *new_hand.cpp*, utilizează pointerul funcției *_new_handler* pentru a cere ca operatorul *new* să apeleze funcția *fara_mem*, atunci când el nu poate satisface o solicitare de alocare a memoriei:

```
#include <iostream.h>
#include <stdlib.h>

extern void (*_new_handler)();
void fara_mem(void)
```

```
  {
    cerr << "Nu mai exista memorie de alocat...\n";
    exit(0);
  }
void main(void)
  {
    _new_handler = fara_mem;
    char *ptr;
    do
      {
        ptr = new char[10000];
        if (ptr)
          cout << "Tocmai a alocat 10000 octeti\n";
      } while (ptr);
  }
```

Observație: *Dacă funcția handler nu poate aloca memorie pentru program, ea trebuie să încheie programul; dacă nu, va apărea o buclă infinită.*

871 STABILIREA UNEI FUNCȚII HANDLER PENTRU NEW CU SET_NEW_HANDLER

C/C++

În secțiunea 870 ați învățat că C++ vă permite să definiți propria dumneavoastră funcție handler pe care programele dumneavoastră o vor apela atunci când operatorul *new* nu va putea satisface solicitările de memorie. Pentru a atribui un handler pentru new, programele atribuie adresa funcției dumneavoastră handler la variabila globală denumită *_new_handler*. Pentru a simplifica procesul de atribuire funcției handler pentru new, multe compilatoare de C++ dispun de o funcție denumită *set_new_handler*, ca mai jos:

```
#include <new.h>
void (* set_new_handler(void (* handler_propriu)()))();
```

Următorul program, *set_newh.cpp*, utilizează funcția *set_new_handler* pentru a instala un handler propriu:

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>
void fara_mem(void)
{
    cerr < "Nu mai exista memorie de alocat...\n";
    exit(0);
}
void main(void)
{
    char *ptr;
    set_new_handler(fara_mem);
    do
```

```
    {
      ptr = new char[10000];
      if (ptr)
        cout < "Tocmai a alocat 10000 octeti\n";
    } while (ptr);
}
```

DETERMINAREA UNEI COMPILĂRI ÎN C++

C/C++ 872

Multe compilatoare de C++ vă permit compilarea programelor în C standard. În funcție de instrucțiunile programelor dumneavoastră, uneori probabil că va trebui să terminați o compilare C standard. Atunci când compilează un program în C++, multe compilatoare de C++ definesc o constantă pe care o puteți testa în cadrul programului. De exemplu, compilatorul de *Turbo C++ Lite* dă valoarea 1 constantei *__cplusplus* atunci când compilează un program în C++. Următorul program, *testcpp.cpp*, utilizează constanta *__cplusplus* pentru a determina dacă se efectuează o compilare C sau C++:

```
#ifdef __cplusplus
#include <iostream.h>
#else
#include <stdio.h>
#endif

void main(void)
  {
#ifdef __cplusplus
   cout << "Compilare C++";
#else
   printf("Compilare C\n");
#endif
  }
```

Compilați programul *testcpp.cpp* ca pe un fișier cu extensia CPP. Apoi, copiați conținutul său într-un fișier cu extensia C și compilați fișierul. Observați procesarea efectuată de compilator.

STRUCTURILE ÎN C++

C/C++ 873

După cum ați învățat, o structură permite programelor dumneavoastră să grupeze informații corelate, de diferite tipuri. Atunci când declarați o structură în C, puteți specifica un nume (tag) cu care declarați mai târziu variabile de tipul structurii:

```
struct tag
  {
    int membru_a;
    float membru_b;
    char membru_c[256];
  };

struct tag variabila_unu, variabila_doi;
```

Când declaraţi o structură în C++, însă, numele structurii devine un tip, cu care programele dumneavoastră pot mai târziu să declare variabile fără să specifice cuvântul cheie *struct*, ca mai jos:

```
struct tag
  {
    int membru_a;
    float membru_b;
    char membru_c[256];
  };
tag variabila_unu, variabila_doi;
```

După cum observaţi, structura în C++ nu solicită cuvântul cheie *struct* înaintea lui tag, la declararea variabilei.

874 INTRODUCEREA FUNCŢIILOR CA MEMBRI AI STRUCTURII

C/C++

Atunci când creaţi programe în C, compilatorul de C vă permite să folosiţi pointeri la funcţii ca membri ai unei structuri:

```
struct tag
  {
    int membru_a;
    int (*membru_b)(); // Pointer la o functie care returneaza int
  };
```

C++ vă permite să optimizaţi conceptul anterior, lăsându-vă să plasaţi efectiv funcţia în cadrul structurii, ca mai jos:

```
struct tag
  {
    int membru_a;
    int membru_b();
  };
```

Când declaraţi funcţia corespunzătoare, aveţi două opţiuni. Aşa cum vom arăta în secţiunea 875, puteţi defini codul funcţiei imediat în cadrul structurii sau îl puteţi defini în afara structurii, cum vom arăta în secţiunea 876. Pentru a apela funcţia, programul dumneavoastră face o simplă referinţă la membrul structurii, ca mai jos:

```
variabila.membru_b(parametri);
```

875 DEFINIREA UNEI FUNCŢII MEMBRU ÎN CADRUL STRUCTURII

C/C++

După cum aţi învăţat, C++ permite programelor dumneavoastră să plaseze funcţii ca membri de structuri. Când structura conţine un membru care este funcţie, puteţi defini codul funcţiei

corespunzătoare în cadrul structurii. Următorul program, *func_mbr.cpp*, defineşte funcţia care corespunde membrului *arata_mesaj*:

```
#include <iostream.h>
struct Msj
  {
    char mesaj[256];
    void arata_mesaj(void) { cout << mesaj; }
  };
void main(void)
  {
    struct Msj carte = { "Jamsa's C/C++ Programmer's Bible" };
    carte.arata_mesaj();
  }
```

Programul *func_mbr.cpp* invocă funcţia membru *arata_mesaj*, care afişează membrul *mesaj*. Aşa cum a arătat codul anterior, programul defineşte funcţia *arata_mesaj* în cadrul structurii. În secţiunea 876, veţi învăţa cum se defineşte funcţia în afara structurii.

DEFINIREA UNEI FUNCŢII MEMBRU ÎN AFARA STRUCTURII

C/C++ 876

După cum aţi învăţat, C++ vă permite plasarea funcţiilor ca membri într-o structură. În secţiunea 875, aţi definit funcţia membru *arata_mesaj* chiar în cadrul structurii. Următorul program, *func_doi.cpp*, defineşte funcţia în afara structurii. Pentru a corespunde funcţia cu structura *Msj*, programul precede numele funcţiei cu numele structurii, urmat de două puncte duble:

```
#include <iostream.h>
struct Msj
  {
    char mesaj [256];
    void arata_mesaj(char *mesaj);
  };
void Msj::arata_mesaj(char *mesaj)
  {
    cout << mesaj;
  }
void main(void)
  {
    struct Msj carte = { "Jamsa's C/C++ Programmer's Bible" };
    carte.arata_mesaj(carte.mesaj);
  }
```

877 TRANSMITEREA PARAMETRILOR CĂTRE O FUNCȚIE MEMBRU

C/C++

După cum ați învățat, C++ vă permite să plasați funcții ca membri ai unei structuri. Secțiunile 875 și 876 au utilizat funcția membru *arata_mesaj* în cadrul structurii *Msj*. Atunci când plasați o funcție ca un membru de structură, puteți trata funcția exact la fel ca orice funcție C++. Cu alte cuvinte, puteți să transmiteți parametri către funcție și să declarați variabile locale în cadrul funcției. Următorul program, *functrei.cpp*, transmite valoarea 1500 către funcția *arata_titlu*:

```
#include <iostream.h>
struct Msj
  {
    char primul[256];
    void arata_titlu(int valoare)
      {
        cout << primul << valoare << " Sectiuni C/C++";
      }
  };
void main(void)
  {
    struct Msj carte = { "Aceasta carte are " };
    carte.arata_titlu(1500);
  }
```

878 MAI MULTE VARIABILE ALE ACELEIAȘI STRUCTURI

C/C++

În secțiunea 875, ați definit o structură care conține o funcție ca membru. Programul *func_mbr.cpp* declară apoi o variabilă de tipul structurii. Următorul program, *multstru.cpp*, declară câteva variabile de tipul *Msj*, atribuie fiecăreia un unic șir de caractere, iar apoi afișează mesajul utilizând funcția *arata_mesaj*:

```
#include <iostream.h>
struct Msj
  {
    char mesaj[256];
    void arata_mesaj(void) { cout << mesaj; }
  };
void main(void)
  {
    struct Msj carte = { "Jamsa's C/C++ Programmer's Bible\n" };
    struct Msj sectiune = { "Introducere in C++" };
    carte.arata_mesaj();
    sectiune.arata_mesaj();
  }
```

STRUCTURI DIFERITE CU ACELEAȘI NUME DE FUNCȚII MEMBRE

C/C++ 879

După cum ați învățat, C++ permite programelor dumneavoastră să plaseze funcții ca membri în cadrul unei structuri. Atunci când programele dumneavoastră utilizează diferite structuri, este posibil ca uneori două structuri să aibă aceleași nume de membri. Următorul program, *un_nume.cpp*, creează două structuri diferite, *Msj* și *MsjMaj*. Ambele structuri utilizează funcția membru *arata_mesaj*. Compilatorul de C++ face diferența între cele două nume de funcții din cele două structuri la fel cum o face în cazul unor funcții care nu sunt membri de structură:

```
#include <iostream.h>
#include <string.h>

struct Msj
  {
  char mesaj[256];
  void arata_mesaj(void) { cout << mesaj; }
  };
struct MsjMaj
  {
    char mesaj[256];
    void arata_mesaj(void) { cout << strupr(mesaj); }
  };
void main(void)
  {
    Msj carte = { "Totul despre C/C++\n" };
    MsjMaj carte_maj = { "TOTUL DESPRE C/C++\n" };
    carte.arata_mesaj();
    carte_maj.arata_mesaj();
  }
```

Structurile din programul *un_nume.cpp* definesc funcțiile în cadrul structurii înseși. În secțiunea 880, însă, veți învăța modul în care se diferențiază funcțiile pe care programele le definesc în afara structurilor.

FUNCȚII DIFERITE CU ACELAȘI NUME DE MEMBRU

C/C++ 880

În secțiunea 879, ați învățat că un compilator de C++ va distinge între funcții membre ale unor tipuri diferite de structuri. Următorul program, *difnume.cpp*, definește funcții membru în afara structurilor cărora le corespund. Pentru a face diferența între funcțiile membru, programul precede fiecare definire a funcțiilor cu numele corespunzător de structură, urmat de două puncte duble:

```
#include <iostream.h>
#include <string.h>

struct Msj
```

```
   {
     char mesaj[256];
     void arata_mesaj(void);
   };
struct MsjMaj
   {
     char mesaj[256];
     void arata_mesaj(void);
   };
void Msj::arata_mesaj(void)
   {
     cout << mesaj;
   }
void MsjMaj::arata_mesaj(void)
   {
     cout << strupr(mesaj);
   }
void main(void)
   {
     Msj carte = { "Totul despre C/C++\n" };
     MsjMaj carte_maj= { "TOTUL DESPRE C/C++\n" };
     carte.arata_mesaj();
     carte_maj.arata_mesaj();
   }
```

881 OBIECTELE

C/C++

În cel mai simplu înțeles, un *obiect* este un lucru sau o entitate din lumea reală. Atunci când programatorii creează programe, ei scriu instrucțiuni care operează cu diferite lucruri, cum ar fi variabile sau fișiere. Obiecte diferite posedă *operații* diferite pe care programele dumneavoastră le efectuează asupra acestor obiecte. De exemplu, fiind dat un obiect *fișier*, programul poate executa operații cum ar fi citirea, scrierea sau tipărirea fișierului. După cum veți învăța, programele în C++ definesc obiecte în termeni de *clase*. O *clasă obiect* (sau pur și simplu, o „clasă") definește datele pe care obiectul le va stoca și funcțiile care vor opera pe aceste date. Programele în C++ se referă adesea la funcțiile care manipulează datele claselor ca la *metode*. De exemplu, majoritatea programelor dumneavoastră în C++ au utilizat deja obiectele *cin* și *cout*. Pentru *cin* și *cout*, fluxul I/O este obiectul, iar funcții ca *cin.get* și *cout.put* sunt *operații* asupra obiectului.

882 PROGRAMAREA ORIENTATĂ PE OBIECTE

C/C++

Pentru programatori, un *obiect* este o colecție de date și un set de operații, numite *metode*, care instrumentează datele. *Programarea orientată pe obiecte* este calea prin care programele sunt gândite în termeni de obiecte (lucruri) care alcătuiesc un sistem. După ce ați identificat obiectele, puteți determina operațiile pe care sistemul le efectuează în mod obișnuit asupra obiectelor. Dacă aveți un obiect *document*, de exemplu, operațiile comune pot cuprinde tipărirea, verificarea ortografiei, transmiterea unui fax sau chiar eliminarea.

Programarea orientată pe obiecte nu solicită un limbaj de programare special, cum ar fi C++. Puteți scrie programe orientate pe obiecte în limbaje cum ar fi COBOL sau FORTRAN. Însă, așa cum veți învăța, limbajele de programare descrise ca „orientate pe obiecte" dispun în mod obișnuit de structuri cu date care permit programelor gruparea datelor și metodelor într-o singură variabilă.

Așa cum veți învăța, programarea pe obiecte are mari avantaje, două dintre ele fiind reutilizarea obiectelor și ușurința înțelegerii. Pe măsură ce scrieți mai multe programe, veți observa că puteți frecvent folosi obiectele scrise, pentru mai multe programe. Decât să construiască o colecție de biblioteci de funcții, programarea orientată pe obiecte construiește *biblioteci de clase.* De asemenea, atunci când creați programe legate de grupuri de obiecte, de datele și metodele lor, dumneavoastră (și cei care vă citesc programul) veți înțelege mai repede programele orientate pe obiecte decât programele care nu sunt orientate pe obiecte (cel puțin, după ce veți învăța sintaxa limbajului de programare pe care îl folosiți). Programatorii și informaticienii denumesc frecvent limbajul C++ ca fiind o extensie orientată pe obiecte a limbajului C. Multe dintre secțiunile care urmează examinează caracteristicile de programare pe obiecte ale limbajului C++.

DE CE TREBUIE SĂ UTILIZĂM OBIECTE



Începând să lucrați în C++, trebuie să înțelegeți de ce programarea orientată pe obiecte este atât de importantă. Există numeroși termeni de inginerie software pe care programatorii îi utilizează frecvent în programarea orientată pe obiecte. Deși inginerii sofware sunt departe de a fi de acord în legătură cu cea mai nimerită utilizare a obiectelor, majoritatea susțin că utilizarea obiectelor oferă următoarele avantaje:

- **Ușurința proiectării și a reutilizării codului** – După ce codul operează corespunzător, utilizarea obiectelor mărește posibilitatea de reutilizare a proiectului sau codului creat pentru o aplicație, într-o altă aplicație.
- **Fiabilitate crescută** – O dată corect testate bibliotecile de obiecte, utilizarea codului existent va mări fiabilitate programelor.
- **Ușurința înțelegerii** – Utilizarea obiectelor îi ajută pe programatori să înțeleagă și să se concentreze asupra elementelor cheie ale sistemului. Utilizarea obiectelor permite proiectanților și programatorilor să se concentreze asupra fragmentelor cele mai mici ale sistemului. Astfel se creează un cadru în care proiectanții se pot concentra mai mult asupra operațiilor executate de program asupra acestor obiecte, asupra informației stocate în aceste obiecte și asupra altor componente cheie ale sistemului.
- **Creșterea abstractizării** – Abstractizarea permite proiectanților și programatorilor „o privire asupra matricei în ansamblu", ignorând temporar detaliile elementare, pentru a opera cu elementele de sistem pe care le înțeleg mai ușor. De exemplu, prin concentrarea numai asupra obiectelor procesorului de text din următoarea secțiune, implementarea procesorului de text va deveni mai puțin descurajantă.
- **Creșterea capacității de încapsulare** – *Încapsularea* (discutată în secțiunea 887) grupează toate părțile unui obiect într-un pachet unic și bine organizat. De exemplu, clasa *Carte* definită anterior, combină funcțiile și câmpurile de date pe care programul le prelucrează pentru o Carte. Programatorii care operează cu clasa *Carte* nu trebuie să cunoască fiecare parte a clasei, ci doar să folosească clasa în cadrul programului. Clasa, aduce la rândul ei, toate elementele necesare.

- **Ascunderea crescândă a informației** – *ascunderea informației* este capacitatea unui program de a trata o funcție, o procedură sau un obiect drept o „cutie neagră", utilizând elementul pentru a efectua o operație specifică, fără cunoașterea a ceea se întâmplă în interior. În capitolul 1, de exemplu, programele foloseau obiecte de flux I/O pentru intrări și ieșiri fără a fi necesar să înțelegeți cum operează fluxurile.

Pe măsură ce examinați diferitele concepte de programare în C++, pe parcursul acestei cărți veți învăța despre corelarea acestor concepte cu definițiile de mai sus.

884 DIVIZAREA PROGRAMELOR ÎN OBIECTE



În cel mai simplu sens, obiectul este un lucru. Câinii, cărțile și calculatoarele sunt toate obiecte. În trecut, programatorii au gândit programele ca o lungă listă de instrucțiuni care execută o anumită sarcină. În schimb, când creați programe orientate pe obiecte, vă ocupați de obiectele care alcătuiesc programul. De exemplu, să presupunem că scrieți un program care implementează un procesor de text simplu. Dacă vă gândiți la toate funcțiile pe care le execută un procesor de texte, vă veți simți curând copleșit. Însă, dacă priviți procesorul de texte ca pe o colecție de obiecte distincte, programul începe să fie mai puțin descurajant. De exemplu, figura 884.1 ilustrează principalele obiecte într-un sistem de procesare de texte.

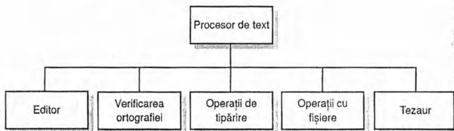


***Figura 884.1** Reprezentarea unui procesor de texte ca o colecție de obiecte.*

Dacă examinați acum fiecare obiect nou, veți observa că acesta, la rândul lui este alcătuit din alte obiecte, cum arătăm în figura 884.2.

Pe măsură ce identificați obiectele pe care le utilizează sistemul, veți observa că multe părți diferite ale programului utilizează aceleași tipuri de obiecte. Prin urmare, atunci când scrieți programe în termeni de obiecte, puteți ușor (și repede) să *reutilizați* codul scris pentru o secțiune, într-o altă secțiune a programului sau poate chiar într-un alt program. Reutilizarea codului este una dintre caracteristicile cele mai puternice ale limbajului C++.

Editor
Document | Ecran | Comenzi
Operațiuni de tipărire
Document | Imprimantă | Comenzi

Verificarea ortografiei
Document | Document | Comenzi
Operații cu fișiere
Document | Fișier | Comenzi
Tezaur
Cuvânt | Document | Comenzi

Figura 884.2 *Identificarea obiectelor suplimentare din cadrul unui procesor de texte.*

După ce ați identificat obiectele, trebuie să determinați scopul fiecărui obiect. Pentru a face aceasta, gândiți-vă la operațiile pe care un obiect le execută sau la operațiile pe care programul le execută asupra obiectului. De exemplu, fiind dat un obiect *fișier*, un program poate să copieze, să șteargă sau să redenumească fișierul. Este important de remarcat că, în general, aceste operații se aplică tuturor fișierelor de pe disc, indiferent de conținutul lor. Aceste operații vor deveni *funcții membre* ale obiectului, pentru care veți scrie mai târziu funcții în C++, în cadrul programului. Identificați, după aceea, informațiile pe care trebuie să le dețineți despre obiect. În cazul unui obiect fișier, trebuie să cunoașteți numele fișierului, dimensiunea, atributele de protecție și, posibil, data și ora la care programul a creat sau a modificat ultima oară fișierul. Aceste elemente de date vor deveni *variabilele membre* ale obiectului fișier. Teoretic, puteți să vedeți obiectul fișier așa cum este prezentat în figura 884.3.

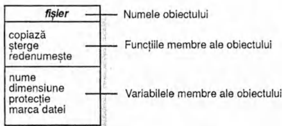


Figura 884.3 *Funcțiile și variabilele membre ale unui fișier obiect.*

885 OBIECTELE ȘI CLASELE

C/C++

Citind articole și cărți despre C++ și programarea orientată pe obiecte, veți întâlni termenii *clasă* și *obiect*. O *clasă* pune la dispoziție un șablon care definește funcțiile membre și datele membre cerute de tipul clasei. Un *obiect*, pe de altă parte, este o *instanță* sau un exemplu specific al unei clase – în principal o variabilă *obiect*. Trebuie să definiți clasa înaintea declarării obiectelor.

Pentru declararea unei variabile obiect, trebuie pur și simplu să specificați tipul clasei, urmat de numele variabilei obiect, ca mai jos:

```
nume_clasa nume_obiect;
```

Programatorii se referă frecvent la procesul de creare a unui obiect cu denumirea *instanțierea unui obiect* sau *crearea unei instanțe obiect*.

886 CLASELE C++

C/C++

De-a lungul acestei cărți, programele au utilizat structuri pentru gruparea datelor corelate. După cum ați învățat, C++ permite programelor pe care le realizați, să utilizeze funcții ca membri de structuri. O *clasă* C++ poate fi reprezentată cel mai bine ca o extindere a conceptului de structură. O clasă, ca și o structură, descrie un șablon pentru viitoare declarări de variabile – nu alocă memorie pentru o variabilă. O clasă are un nume (tag) și câmpuri membre. Următoarea definire, de exemplu, ilustrează o clasă simplă, denumită *Carte*:

```
class Carte
  {
    public:
      char titlu[256];
      char autor[64];
      float pret;
      void arata_titlu(void) { cout << titlu << '\n' };
      float da_pret(void) { return(pret); };
  };
```

După cum puteți observa, definirea clasei este foarte asemănătoare cu cea a unei structuri. Unicul element nou este eticheta *public*. Secțiunea 896 va discuta scopul etichetei *public*. Următorul program, *primclas.cpp*, utilizează clasa *Carte* pentru a afișa informații despre o carte:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte
  {
    public:
      char titlu[256];
      char autor[64];
      float pret;
```

```
        void arata_titlu(void) { cout << titlu << '\n'; };
        float da_pret(void) { return(pret); };
    };
void main(void)
    {
      Carte capitole;

      strcpy(capitole.titlu, "Jamsa's C/C++ Programmer's Bible");
      strcpy(capitole.autor, "Jamsa si Klander");
      capitole.pret = 49.95;
      capitole.arata_titlu();
      cout << "Pretul carti este " << setprecision(2) <<
           capitole.da_pret();
    }
```

ÎNCAPSULAREA

C/C++ 887

Citind articole și cărți despre programarea orientată pe obiecte și C++, puteți întâlni termenul de *încapsulare*. În cel mai simplu sens, încapsularea este combinarea de date și metode într-o singură structură de date. Încapsularea grupează împreună toate componentele unui obiect. În sensul „orientării pe obiecte", încapsularea definește, de asemenea, modalitatea în care obiectele însele și restul programului se pot referi la datele obiectului. După cum ați învățat, clasele C++ permit separarea datelor în secțiuni publice și private. Programele pot accesa datele unui obiect privat numai utilizând metode definite public. Gruparea împreună a datelor obiectelor și împărțirea datelor în secțiuni publice și private protejează datele față de utilizarea lor eronată în programe. În C++, clasele reprezintă instrumentul fundamental de încapsulare.

POLIMORFISMUL

C/C++ 888

Citind articole și cărți despre C++, veți întâlni frecvent termenul *polimorfism*. Polimorfismul permite programelor să aplice aceeași operație la obiecte de tipuri diferite. Deoarece polimorfismul permite programatorilor aplicarea aceleiași operații pentru mai multe tipuri, polimorfismul permite utilizarea aceleiași interfețe de acces la obiecte diferite. În C++, accesul la polimorfism este furnizat de *funcțiile virtuale*. În cel mai simplu înțeles, o funcție virtuală este un pointer către o funcție pe care compilatorul o rezolvă în cursul execuției. În raport de funcția către care indică funcția virtuală, operația efectuată de program va prezenta diferențe. În consecință, o singură interfață (funcția virtuală) poate furniza accesul la diferite operații. Secțiunea 1090 prezintă funcțiile virtuale în detaliu.

MOȘTENIREA

C/C++ 889

Atunci când derivați clase utilizând mecanismul moștenirii din C++, desenarea unei imagini vă poate facilita înțelegerea relațiilor dintre clase. Veți observa că o clasă pe care o derivați dintr-una sau mai multe clase, poate deveni al rândul ei clasa de bază pentru alte clase. Când definiți propriile dumneavoastră clase, începeți cu caracteristicile generale, iar la derivarea unei noi clase mergeți spre caracteristicile specifice. De exemplu, dacă derivați clase pentru

tipuri de câini, prima dumneavoastră clasă de bază poate fi pur și simplu *Caini*. Clasa de bază *Caini* trebuie să conțină caracteristicile comune tuturor raselor de câini, cum ar fi nume, origine, înălțime, greutate și culoare. Următorul nivel de creare a claselor poate deveni mai rafinat. Al doilea nivel de tipuri de clasă, *CainiCuPete* și *CainiFaraPete*, de exemplu, ar trebui să moștenească toate caracteristicile comune pe care le-ați definit în clasa de bază *Caini*. Apoi, pentru a detalia pedigriul (de exemplu, între dalmațieni și labradori), puteți utiliza acest nivel secund de clase, drept clase de bază pentru alte definiri de clase. Nivelurile de clase de bază vor crește, teoretic în mod similar cu un arbore genealogic, cum arătăm în figura 889.

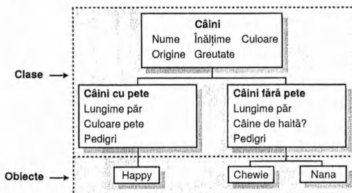


Figura 889 *Arborele moștenirii clasei* ***Caini.***

890 DECIZIA ÎNTRE CLASE ȘI STRUCTURI

C/C++

Anterior în cursul acestei cărți, ați învățat despre structurile C. Așa cum ați învățat despre clase, ar trebui să recunoașteți că sintaxa în cazul operării cu clase este foarte asemănătoare cu sintaxa utilizată la structurile din C. Poate că vă veți mira că trebuie să utilizați clase, în locul structurilor sau chiar al uniunilor. După cum știți, clasele, structurile și uniunile permit programelor dumneavoastră să stocheze date corelate. Programele dumneavoastră ar trebui să utilizeze clase oriunde programul efectuează operații specifice asupra datelor. Să considerăm, de exemplu, că dacă aveți doar nevoie să stocați o dată calendaristică, puteți utiliza o structură sau o uniune. Însă, dacă doriți ca programul să formateze și să afișeze data calendaristică, să ordoneze data sau să compare două date, ar trebui să utilizați o clasă. De asemenea, dacă trebuie să alegeți între utilizarea unei structuri sau a unei uniuni, ar trebui să fundamentați decizia dumneavoastră pe numărul de valori pe care structura de date trebuie să le stocheze la un moment dat. În sfârșit, țineți seama de faptul că, în mod implicit, membrii claselor sunt privați, iar membrii structurilor și uniunilor sunt publici.

Dacă studiați structurile din C++, veți observa că ele acceptă multe caracteristici identice cu clasele din C++, cum ar fi date publice și private, funcții membre și așa mai departe. De regulă, când creați obiecte, utilizați clase.

CREAREA UNUI MODEL SIMPLU DE CLASĂ

C/C++ 891

Cea mai bună metodă de înţelegere a claselor C++ şi a obiectelor este crearea unui program simplu. În următoarea secţiune, exemplul de program, *filme.cpp*, creează o clasă denumită *film*. Apoi el creează două obiecte de tip *film*, numite *fugar* şi *neobosit*. Programul defineşte clasa *film*, ca mai jos:

```
class film
  {
    public:
      char nume[64];
      char prim_actor[64];
      char aldoilea_actor[64];
      void arata_film(void);
      void initializare(char *nume, char *prim, char *aldoilea);
  };
```

După cum vedeţi, clasa *film* utilizează trei variabile membre şi două funcţii membre. În continuarea definirii clasei, programul trebuie să definească funcţiile membre *arata_film* şi *initializare*, ca mai jos:

```
void film::arata_film(void)
  {
    cout << "Numele filmului: " << nume << endl;
    cout << "Interpreteaza: " << prim_actor << " si " <<
               aldoilea_actor << endl << endl;
  }
void film::initializare(char *nume_film, char *prim, char *aldoilea)
  {
    strcpy(nume, nume_film);
    strcpy(prim_actor, prim);
    strcpy(aldoilea_actor, aldoilea);
  }
```

Definirea funcţiilor clasei este foarte asemănătoare cu definiţia standard a unei funcţii. Însă, există două diferenţe principale. Prima, numele clasei şi două puncte duble preced numele funcţiei:

```
void film::initializare(char *nume_film, char *prim, char *aldoilea)
```

A doua diferenţă, în cadrul funcţiei unei clase, instrucţiunile pot face referinţă direct la variabilele membre ale clasei, ca mai jos:

```
void film::initializare(char *nume_film, char *prim, char *aldoilea)
  {
    strcpy(nume, nume_film);
    strcpy(prim_actor, prim);
    strcpy(aldoilea_actor, aldoilea);
  }
```

892 IMPLEMENTAREA UNUI PROGRAM SIMPLU PENTRU CREAREA UNEI CLASE

C/C++

În secțiunea 891, ați învățat despre componentele unei clase simple, *film*. Acum, pentru că ați înțeles cum se creează o clasă, trebuie să implementați clasa pentru a înțelege mai bine cum veți lucra cu clasele în cadrul programelor dumneavoastră. Programul *filme.cpp* implementează clasa *film*:

```
#include <iostream.h>
#include <string.h>

class film
  {
    public:
      char nume[64];
      char prim_actor[64];
      char aldoilea_actor[64];
      void arata_film(void);
      void initializare(char *nume, char *prim, char *aldoilea);
  };
void film::arata_film(void)
  {
    cout << "Numele filmului: " << nume << endl;
    cout << "Interpreteaza: " << prim_actor << " si " <<
              aldoilea_actor << endl << endl;
  }
void film::initializare(char *nume-film, char *prim, char *aldoilea)
  {
    strcpy(nume, nume-film);
    strcpy(prim_actor, prim);
    strcpy(aldoilea_actor, aldoilea);
  }
void main(void)
  {
    film fugar, neobosit;
    fugar.initializare("Fugarul", "Harrison Ford", "Tommy Lee
                         Jones");
    neobosit.initializare("Nopti albe in Seattle", "Tom Hanks",
                         "Meg Ryan");
    fugar.arata_film();
    neobosit.arata_film();
  }
```

După cum vedeți, programul creează două obiecte de tip *film*:

```
film fugar, neobosit;
```

În programul *filme.cpp*, programul utilizează funcția membru *initializare* pentru a inițializa variabilele membre ale clasei. În secțiunile următoare, veți învăța modul în care se folosește funcția *constructor* pentru a inițializa variabilele membre într-un mod mai firesc.

DEFINIREA COMPONENTELOR UNEI CLASE

C/C++ 893

După cum aţi învăţat, o clasă constă într-unul sau mai multe componente distincte, care pot fi variabile, funcţii sau ambele. *Declaraţia unei clase* defineşte un nou tip de clasă care leagă codul şi datele. Programele dumneavoastră vor utiliza noul tip pentru a declara obiecte ale acestei clase. Deci, o clasă reprezintă o abstractizare logică, iar un obiect are o existenţă fizică. Cu alte cuvinte, un obiect este o *instanţă* a unei clase.

Aşa cum aţi văzut în secţiunile precedente, declararea unei clase este similară din punctul de vedere al sintaxei, cu declararea unei structuri. Următorul cod arată forma generală a unei clase:

```
class nume_clasa
  {
    date si functii private
specificator-acces:
    date si functii
specificator-acces:
    date si functii
.
.
.
specificator-acces:
    date si functii
  }lista-obiecte;
```

Lista_obiecte este opţională. Dacă este prezentă, ea declară obiecte ale clasei. *Specificator-acces* este unul din cele trei cuvinte cheie utilizate la definirea claselor în C++ pe care le-aţi învăţat anterior: *public, private* şi *protected*.

Datele şi funcţiile publice, pe care le conţine clasa, sunt cunoscute de regulă ca *proprietăţi* şi *metode*. După cum aţi învăţat, metodele sunt, de asemenea, cunoscute ca *funcţii de interfaţă*.

OPERATORUL DE REZOLUŢIE A DOMENIULUI DE VALABILITATE

C/C++ 894

După cum aţi învăţat, programele dumneavoastră vor utiliza *operatorul de rezoluţie a domeniului de valabilitate* (operatorul ::) pentru a lega numele clasei de un nume de membru, pentru a spune compilatorului cărei clase îi aparţine respectivul membru. Operatorul de rezoluţie a domeniului poate, de asemenea, să permită programelor dumneavoastră să acceseze un nume dintr-un domeniu închis pe care îl ascunde o declaraţie locală cu acelaşi nume. De exemplu, să considerăm următorul fragment de cod:

```
// Instructiuni de program
//
int i; // global
void f();
  int i; // local
```

```
   i = 10; // se refera la i local
//
// Instructiuni de program
```

Însă, dacă funcţia *f* cere o referinţă la *i* global şi nu la *i* local, puteţi rescrie fragmentul:

```
// Instructiuni de program
//
int i; // global
void f();
   int i; // local
   ::i = 10; // se refera la i global, nu i local
//
// Instructiuni de program
```

895 UTILIZAREA SAU OMITEREA NUMELUI CLASEI ÎN DECLARAŢII

C/C++

Aşa cum aţi învăţat, o clasă defineşte un şablon pentru viitoare declarări de variabile. După ce definiţi o clasă, programul dumneavoastră poate declara o clasă într-una dintre cele două modalităţi posibile: programul poate utiliza clasa însăşi sau el poate pur şi simplu să specifice numele clasei (tag):

```
class Carte
   {
     public:
       char titlu[256];
       char autor[64];
       float pret;
       void arata_titlu(void) { cout << titlu << '\n' };
       float da_pret(void) { return(pret); };
   };
// Declara variabile de tipul clasei
class Carte capitole;
Carte jurnal;
```

După cum vedeţi, C++ utilizează numele clasei în aceeaşi manieră cum se utilizează în cazul structurilor: pentru a crea un tip cu care puteţi mai târziu să declaraţi alte variabile.

896 ETICHETA PUBLIC:

C/C++

În secţiunea 895 aţi creat o clasă simplă, numită Carte, care conţine eticheta *public*:

```
class Carte
   {
     public:
       char titlu[256];
       char autor[64];
```

```
    float pret;
    void arata_titlu(void) { cout << titlu << '\n' };
    float da_pret(void) { return(pret); };
 };
```

Spre deosebire de o structură, ai cărei membri sunt toţi accesibili programului, o clasă poate avea membri pe care programul îi poate accesa direct utilizând operatorul *punct* şi alţi membri (denumiţi *membri privaţi*) pe care programul nu îi poate accesa direct. Eticheta *public:* identifică membrii clasei pe care programul îi poate utiliza cu operatorul *punct*. Dacă doriţi ca programul să acceseze un anumit membru direct, trebuie să declaraţi respectivul membru în cadrul membrilor publici ai clasei.

ASCUNDEREA INFORMAŢIILOR

C/C++ 897

Ascunderea informaţiilor este procesul de ascundere a unor detalii de implementare elementare ale unei funcţii, program sau clase. Ascunderea informaţiilor permite programatorilor tratarea funcţiilor şi claselor drept *cutii negre*. Cu alte cuvinte, dacă programatorul transmite o valoare către o funcţie, el ştie rezultatul specific care va apărea. Programatorul nu are nevoie să ştie cum calculează funcţia acel rezultat, ci numai că funcţia lucrează. De exemplu, majoritatea programatorilor nu cunosc suportul matematic din spatele funcţiei *tanh*, care returnează tangenta hiperbolică a unui unghi. El ştie, însă, că dacă transmite o anumită valoare către funcţie, rezultatul ştiut va apărea. Pentru a utiliza funcţia, programatorul trebuie să ştie numai parametrii de intrare şi valorile pe care funcţia le returnează.

În programarea orientată pe obiecte, un obiect poate avea detalii de implementare fundamentale. De exemplu, Microsoft *Word®*, *Excel®* sau alte programe pot stoca date într-un *document*. Pentru a utiliza obiectul *document*, programul nu trebuie să cunoască formatul. În schimb, programul trebuie să efectueze operaţii de citire, scriere, tipărire şi transmitere de fax fără să cunoască detalii ale obiectului. Pentru a ajuta programatorul să ascundă detalii elementare ale unui obiect, C++ permite separarea definiţiei clasei în părţi private şi publice. Programul poate accesa direct datele şi metodele publice, dar nu poate accesa direct datele şi metodele private.

ETICHETA PRIVATE:

C/C++ 898

După cum aţi învăţat, C++ vă permite separarea definiţiei clasei în secţiuni publice şi private. Programul poate folosi operatorul *punct* pentru a accesa datele şi metodele publice. Însă el nu poate utiliza operatorul *punct* pentru a accesa direct datele şi metodele private. Următoarea definiţie de clasă extinde clasa *Carte* pentru a include date şi metode publice şi private:

```
class Carte
 {
   public:
    char titlu[256];
    char autor[64];
    float pret;
    void arata_titlu(void) { cout << titlu << '\n' };
    float da_pret(void) { return(pret); };
```

```
    void arata_carte(void)
      {
        arata_titlu();
        arata_editura();
      };
    void atrib_editura(char *nume) { strcpy(editura, nume); };
  private:
    char editura[256];
    void arata_editura(void) { cout << editura << '\n'; };
};
```

Programul poate utiliza operatorul *punct* pentru a accesa direct date și metode care se situează în secțiunea publică. Unica modalitate de accesare a datelor și metodelor private este prin intermediul metodelor publice. Secțiunea 900 prezintă un program care lucrează atât cu date publice, cât și private.

899 UTILIZAREA ETICHETEI PROTECTED: C/C++

Așa cum ați învățat, C++ vă permite clasificarea membrilor unei clase în publici și privați. Clasificarea membrilor în publici și privați controlează modalitatea în care programul are acces la membrii clasei. Atunci când utilizați moștenirea pentru a deriva o clasă dintr-o alta, C++ adaugă o a treia categorie de membri: *protejați (protected)*. Un *membru protejat* are un statut intermediar între membri privați și cei publici. Pentru clasa de bază, obiectele derivate pot accesa membri protejați, ca și cum ar fi publici. În afara obiectelor derivate, însă, numai rutinele de interfață publice pot accesa membri protejați. Următorul cod adaugă doi membri protejați la definiția clasei *Carte*.

```
class Carte
  {
  public:
    Carte(char *titlu) { strcpy(Carte::titlu, titlu); };
    void arata_titlu(void) { cout << titlu << endl; };
  protected:
    float cost;
    void arata_cost(void) { cout << cost << endl; };
  private:
    char titlu[64];
  };
```

În exemplul precedent, obiectele derivate din clasa *Carte* pot accesa membri *cost* și *arata_cost* ca și cum ar fi publici. În afară de obiectele derivate, însă, programul trebuie să trateze membrii protejați ca și cum ar fi privați.

900 UTILIZAREA DATELOR PUBLICE ȘI PRIVATE C/C++

După cum ați învățat, C++ vă permite separarea definiției clasei în date și metode publice și private. Programele pot accesa datele și metodele publice utilizând operatorul *punct*. Pentru a accesa datele și metodele private, însă, programul trebuie să apeleze metode publice. Programul nu poate manipula sau invoca direct date și metode private. Următorul program, *pub_priv.cpp*, ilustrează utilizarea datelor publice și private:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte
  {
    public:
      char titlu[256];
      char autor[64];
      float pret;
      void arata_titlu(void) { cout << titlu << '\n'; };
      float da_pret(void) { return(pret); };
      void arata_carte(void)
        {
          arata_titlu();
          arata_editura();
        };
      void atrib_editura(char *nume) { strcpy(editura, nume); };
    private:
      char editura[256];
      void arata_editura(void) { cout << editura << '\n'; };
  };
void main(void)
  {
    Carte bible;

    strcpy(bible.titlu, "Jamsa's C/C++ Programmer's Bible");
    strcpy(bible.autor, "Jamsa si Klander");
    bible.pret = 49.95;
    bible.atrib_editura("Jamsa Press");
    bible.arata_carte();
  }
```

După cum vedeți, metoda publică *atrib_editura* inițializează membrul privat *editura*. Dacă programul ar încerca să acceseze direct membrul *editura*, compilatorul ar fi generat o eroare. În mod similar, programul utilizează metoda publică *arata_carte*, care la rândul ei invocă metoda privată *arata_editura*. Din nou, programul nu poate accesa direct o metodă privată.

CE ASCUNDEM ȘI CE FACEM PUBLIC

C/C++ 901

După cum ați învățat, C++ vă permite separarea definiției clasei în secțiuni publice și private. Una dintre cele mai dificile probleme pe care le întâmpină programatorii neexperimentați în programarea orientată pe obiecte este determinarea acelor membri ai fiecărei clase pe care ar trebui să îi ascundă și pe cei pe care ar trebui să îi facă publici. Ca regulă generală, cu cât un program știe mai puțin despre clase, cu atât mai bine. De aceea, ar trebui să încercați să utilizați date și metode private cât mai des posibil. Atunci când utilizați date și metode private, programele care utilizează obiectul trebuie să folosească metodele publice ale obiectului pentru accesul la datele obiectului. Așa cum veți învăța în secțiunea 902, obligând

programul să lucreze cu datele obiectului utilizând numai metode publice, se poate micșora numărul erorilor de programare. Cu alte cuvinte, de obicei, nu doriți ca programul să lucreze direct cu datele obiectului, utilizând doar operatorul *punct*. Apelând la această manieră de utilizare a datelor private, optimizați ascunderea informațiilor.

902 METODELE PUBLICE SUNT DESEORI FUNCȚII DE INTERFAȚĂ

C/C++

După cum ați învățat în secțiunea 901, programele dumneavoastră este bine să plaseze majoritatea datelor unui obiect în secțiunea privată a definiției clasei. Atunci când programele plasează datele despre obiect în secțiunea privată, celelalte programe nu pot accesa datele decât prin apelul la metodele publice din clasa respectivă. În acest mod, metodele publice pun la dispoziția programului o interfață la datele obiectului. Utilizând aceste funcții de interfață, programele dumneavoastră pot verifica dacă valoarea atribuită unui membru este validă. De exemplu, să presupunem că membrul în clasa *ReactorNuclear* trebuie să conțină numai valorile de la 1 la 5. Dacă membrul este public, programul poate aloca o valoare nevalidă utilizând operatorul *punct*, ca mai jos:

```
radiatie.topire_miez = 99;
```

Prin restrângerea accesului la membrul *topire_miez* la metoda publică *proc_topire_miez*, obiectul poate verifica valoarea, cum prezentăm mai jos:

```
int proc_topire_miez(int valoare)
  {
    if ((valoare >= 1) && (valoare <= 5))
      {
        radiatie.topire_miez = valoare;
        return(0);
      }
    else
      return(-1); // Valoare nevalida
  }
```

Prin restrângerea accesului datelor obiectului la metodele publice, singurele operații pe care programul le poate efectua asupra datelor din cadrul obiectului sunt operațiile pe care le definește obiectul însuși.

903 DEFINIREA FUNCȚIILOR CLASEI ÎN AFARA CLASEI

C/C++

Mai multe dintre secțiunile precedente au creat clase simple care definesc membri funcții chiar în cadrul claselor. Pe măsură ce dimensiunea funcțiilor din clasele dumneavoastră crește, în cele din urmă veți defini funcțiile în afara clasei. Următorul program, *cartefct.cpp*, definește funcțiile pentru obiectul *Carte* în afara clasei. După cum veți vedea, programul identifică funcțiile clasei prin precedarea fiecărui nume al funcțiilor cu numele clasei și două puncte duble:

```
#include <iostream.h>
#include <iomanip.h>
```

```
#include <string.h>
class Carte
  {
    public:
      char titlu[256];
      char autor[64];
      float pret;
      void arata_titlu(void);
      float da_pret(void);
      void arata_carte(void);
      void atrib_editura(char *nume);
    private:
      char editura[256];
      void arata_editura(void);
  };
void Carte::arata_titlu(void)
  { cout << titlu << '\n'; };
float Carte::da_pret(void)
  { return(pret); };
void Carte::arata_carte(void)
  {
    arata_titlu();
    arata_editura();
  };
void Carte::atrib_editura(char *nume)
  { strcpy(editura, nume); };
void Carte::arata_editura(void)
  { cout << editura << '\n'; };
void main(void)
  {
    Carte capitole;
    strcpy(capitole.titlu, "Jamsa's C/C++ Programmer's Bible");
    strcpy(capitole.autor, "Jamsa si Klander");
    capitole.pret = 49.95;
    capitole.atrib_editura("Jamsa Press");
    capitole.arata_carte();
  }
```

DEFINIREA METODELOR ÎN INTERIORUL ȘI ÎN EXTERIORUL CLASELOR

C/C++ 904

După cum ați învățat, C++ vă permite definirea metodelor în interiorul și în exteriorul declarației claselor. Decizia pe care o luați în legătură cu plasarea definirii metodelor afectează codul pe care compilatorul îl creează pentru program. Atunci când definiți o metodă în cadrul clasei, compilatorul va trata fiecare referință la metodă ca un apel la o

funcție *inline*, plasând instrucțiunile corespunzătoare funcției în codul obiect la fiecare referință la metodă. Așa cum ați învățat, utilizând codul *inline* puteți îmbunătăți performanța programului, dar puteți, de asemenea, mări dimensiunile programului. Pe de altă parte, atunci când definiți o funcție în afara clasei, compilatorul nu utilizează cod *inline*. În schimb, el va genera cod pentru o funcție pe care programul o va apela la fiecare referință la metodă. De aceea, când clasa dumneavoastră conține o operație comună și de mici dimensiuni, puteți cere compilatorului să genereze cod *inline* pentru acea metodă. Dacă metoda este mai mare, nu indicați compilatorului să genereze cod *inline*.

905 INSTANȚELE OBIECT

C/C++

Multe cărți de C++ și articole se referă la *instanțe obiect*. Pe scurt, o instanță obiect este o variabilă obiect. După cum ați învățat, o clasă definește un șablon pentru viitoare declarații de variabile. Când, ulterior, declarați un obiect, creați o instanță obiect. Cu alte cuvinte, atunci când compilatorul alocă memorie pentru variabilă, programul creează o instanță obiect. Toate instanțele aceleiași clase au aceleași caracteristici. Pentru intențiile acestei cărți, o instanță este o variabilă a unei anumite clase.

906 INSTANȚELE OBIECT TREBUIE SĂ PARTAJEZE CODUL

C/C++

După cum ați învățat, C++ vă permite definirea metodelor unei clase în cadrul clasei sau în afara ei. Atunci când declarați metodele unei clase în afara clasei, instanțele partajează aceeași copie a metodelor. Dacă, de exemplu, aveți o clasă cu trei metode și creați 100 de instanțe ale acestei clase, programul dumneavoastră va conține numai trei metode. Dacă includeți cod *inline*, însă, instanțele nu vor partaja codul. De aceea, ar trebui să rezervați codul *inline* pentru efectuarea de operații de mici dimensiuni, obișnuite, unde performanța operației este mai importantă decât dimensiunea programului. De exemplu, următorul program, *partaj.cpp*, creează două instanțe ale clasei *Carte*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte
  {
   public:
     char titlu[256];
     char autor[64];
     float pret;
     void arata_titlu(void) { cout << titlu << '\n'; };
     float da_pret(void) { return(pret); };
     void arata_carte(void);
     void atrib_editura(char *nume) { strcpy(editura, nume); };
   private:
     char editura[256];
     void arata_editura(void) { cout << editura << '\n'; };
  };
void Carte::arata_carte(void)
```

```
{
    arata_titlu();
    arata_editura();
};
void main(void)
{
    Carte capitole, jurnal;
    strcpy(capitole.titlu, "Jamsa's C/C++ Programmer's Bible");
    strcpy(capitole.autor, "Jamsa si Klander");
    capitole.pret = 49.95;
    capitole.atrib_editura("Jamsa Press");
    strcpy(jurnal.titlu, "All My Secrets...");
    strcpy(jurnal.autor, "Kris Jamsa");
    jurnal.pret = 9.95;
    jurnal.atrib_editura("Fara");
    capitole.arata_carte();
    jurnal.arata_carte();
}
```

Când compilați programul *partaj.cpp* utilizând *Turbo C++ Lite* pentru a produce un cod în limbaj de asamblare, veți observa că instanțele nu partajează codul pentru metoda *inline*, dar partajează codul definit în afara clasei.

ACCESUL LA MEMBRII CLASEI

C/C++907

În secțiunea precedentă, ați utilizat operatorul *punct* pentru a invoca funcții membre ale unei clase. Atunci când programele plasează obiecte membre după eticheta *public*, programele pot accesa membrii utilizând operatori *punct*. De exemplu, în secțiunea 892, ați creat o clasă simplă *film* și ați accesat funcția sa membru *arata_film*. Următorul program, *public.cpp*, utilizează funcția *initializare* pentru a atribui valori membrilor obiectelor *fugar* și *neobosit*. Programul afișează apoi diferitele valori ale membrului prin referința la membru, utilizând operatorul punct, ca mai jos:

```
#include <iostream.h>
#include <string.h>
class film
{
    public:
      char nume[64];
      char primul_actor[64];
      char aldoilea_actor[64];
      void arata_film(void);
      void initializare(char *nume, char *primul, char *aldoilea);
};
void film::arata_film(void)
```

```
   {
     cout << "Numele filmului: " << nume << endl;
     cout << "Interpreteaza: " << primul_actor << " si " <<
           aldoilea_actor << endl << endl;
   }
void film::initializare(char *film_nume, char *primul,
  char *aldoilea)
   {
     strcpy(nume, film_nume);
     strcpy(primul_actor, primul);
     strcpy(aldoilea_actor, aldoilea);
   }
void main(void)
   {
     film fugar, neobosit;
     fugar.initializare("Fugarul", "Harrison Ford", "Tommy Lee
                     Jones");
     neobosit.initializare("Nopti albe in Seattle", "Tom Hanks",
                "        Meg Ryan");
     cout << "Ultimele doua filme pe care le-am vazut sunt: " <<
           fugar.nume << " si " << nopti albe.nume << endl;
     cout << "Cred ca " << fugar.primul_actor << " a fost mare!"
       << endl;
   }
```

Deoarece membrii clasei sunt publici, programele pot accesa direct acești membri. Când compilați și executați programul *public.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Ultimele doua filme pe care le-am vazut sunt: Fugarul si
  Nopti albe in Seattle
Cred ca Harrison Ford a fost mare!
C:\>
```

Atunci când o clasă definește variabile membre ca fiind publice, programele dumneavoastră le pot accesa utilizând operatori punct. Totuși, așa cum veți învăța în secțiunile următoare, acest acces direct la o variabilă membru nu este întotdeauna de preferat.

908 DIN NOU DESPRE OPERATORUL DE REZOLUȚIE GLOBALĂ



După cum ați învățat, C++ vă permite repetarea numelui funcțiilor și variabilelor în mai multe clase. Cu alte cuvinte, existența funcției *ziua_urmatoare* în clasa *saptamana* nu exclude existența sa în cadrul clasei *luna*. În cadrul programelor dumneavoastră, puteți utiliza *operatorul de rezoluție globală* (::) pentru a preveni confuzia între date și numele funcțiilor. Atunci când trebuie să vă referiți la un membru al unei clase (date sau funcție), precedați numele membrului cu numele clasei și două puncte duble:

```
void film::initializare(char *nume, char *primul_actor,
  char *aldoilea_actor)
   {
```

```
    strcpy(film::nume, nume);
    strcpy(film::primul_actor, primul_actor);
    strcpy(film::aldoilea_actor, aldoilea_actor);
  }
```

INIȚIALIZAREA VALORILOR CLASEI

C/C++909

Așa cum ați văzut în secțiunile anterioare, este important să inițializați valorile în cadrul claselor dumneavoastră de fiecare dată când creați o nouă instanță dintr-o clasă. C++ dispune de mai multe modalități de inițializare a valorilor în cadrul claselor. În următoarele secțiuni veți învăța despre funcțiile constructor, pe care programatorii de C++ le utilizează de obicei pentru a inițializa instanțele noi ale unei clase. Puteți însă, să creați o funcție membru pentru a inițializa o clasă, ca mai jos:

```
void film::initializare(char *nume, char *primul_actor,
  char *aldoilea_actor)
  {
    strcpy(film::nume, nume);
    strcpy(film::primul_actor, primul_actor);
    strcpy(film::aldoilea_actor, aldoilea_actor);
  }
```

UTILIZAREA ALTEI METODE DE INIȚIALIZARE A VALORILOR CLASEI

C/C++910

După cum ați învățat, puteți inițializa valorile unei clase în cadrul unei funcții de inițializare. Așa cum veți învăța în secțiunile următoare, puteți de asemenea să inițializați clasele dumneavoastră în cadrul funcțiilor constructor. Dacă însă examinați programe în C++, probabil veți întâlni o unică tehnică de inițializare a membrilor clasei. Să presupunem, de exemplu, că doriți constructorul *contor* să inițializeze variabila *numar* la 0, ca mai jos:

```
contor::contor(void)
  {
    numar = 0;
    // Alte instructiuni
  }
```

C++ permite nu numai crearea unei funcții de inițializare pentru fiecare clasă, ci de asemenea, permite inițializarea variabilelor membre ale unei clase prin plasarea numelui variabilei precedat de două puncte și a valorii sale înaintea instrucțiunilor funcției, ca mai jos:

```
contor::contor(void) : numar(0)
  {
    // Alte instructiuni
  }
```

Următorul program, *nr_init.cpp*, utilizează formatul constructor pentru inițializarea a trei variabile membre cu valorile 1, 2 și 3:

```
#include <iostream.h>
class obiect
  {
    public:
      obiect::obiect(void);
      void arata_obiect(void);
    private:
      int a;
      int b;
      int c;
  };
obiect::obiect(void) : a(1), b(2), c(3) { };
void obiect::arata_obiect(void)
  {
    cout << "a contine: " << a << endl;
    cout << "b contine: " << b << endl;
    cout << "c contine: " << c << endl;
  }
void main(void)
  {
    obiect numere;
    numere.arata_obiect();
  }
```

Atunci când compilaţi şi executaţi programul *nr_init.cpp*, ecranul dumneavoastră va afişa următoarele:

```
a contine 1
b contine 2
c contine 3
C:\>
```

911 MEMBRII STATICI AI CLASELOR



În cadrul claselor C++, puteţi defini atât date membri, cât şi funcţii ca statice (*static*). După cum veţi învăţa, declararea unui membru de date sau a unei funcţii ca *static* are implicaţii importante asupra claselor C++. C++ gestionează membri de date şi funcţii statice cu reguli diferite decât alte celorlalţi membri şi funcţii. De exemplu, o funcţie *statică* poate să acceseze numai un alt membru *static* în cadrul aceleiaşi clase (la fel şi funcţiile globale şi datele membre). Înainte de a utiliza membri statici în cadrul claselor dumneavoastră, este important să înţelegeţi implicaţiile utilizării lor. Secţiunile 912 şi 913 abordează în detaliu membrii de date şi funcţii statice.

912 UTILIZAREA DATELOR MEMBRE STATICE



După cum aţi învăţat în secţiunea 911, compilatorul de C++ tratează datele membru care sunt precedate de cuvântul cheie *static* diferit faţă de datele membru normale. De fapt, atunci când precedaţi o declaraţie a unei variabile membru cu cuvântul cheie *static*, indicaţi

compilatorului că va exista numai o singură copie a acestei variabile și că toate obiectele clasei vor partaja variabila respectivă. Spre deosebire de membrii obișnuiți, programul nu creează copii individuale ale membrilor *statici* pentru fiecare obiect. Oricât de multe obiecte creează programul, va exista numai o singură copie a fiecărui membru static. De aceea, toate obiectele acelei clase utilizează aceeași variabilă. Compilatorul inițializează toate variabilele statice la zero atunci când programul creează prima instanță obiect.

Când declarați date membre ca statice în cadrul unei clase, nu *definiți* acel membru. Cu alte cuvinte, nu alocați memorie pentru stocarea acelui membru. În schimb, trebuie să dați o definiție globală pentru datele membre statice, undeva în afara clasei. Pentru a furniza o definiție globală, veți redeclara datele membre statice utilizând operatorul de rezoluție globală. Astfel, veți cere compilatorului să aloce memorie pentru stocarea membrului static. Pentru a înțelege mai bine utilizarea și efectele datelor membre statice, studiați următorul program, *stat_mem.cpp*:

```
#include <iostream.h>
class partajata
   {
     static int a;
      int b;
     public:
      void set(int i, int j)
        {
          a=i;
          b=j;
        }
      void arata();
   };
int partajata::a;    //Defineste variabila globala
void partajata::arata()
   {
     cout << "Acesta este static a: " << a << endl;
     cout << "Acesta este non-static b: " << b << endl;
   }
void main(void)
   {
     partajata x, y;
     x.set(1,1);
     x.arata();
     y.set(2,2);
     y.arata();
     x.arata();
   }
```

Când compilați și executați programul *stat_mem.cpp*, el va genera următorul rezultat:

```
Acesta este static a: 1
Acesta este non-static b: 1
Acesta este static a: 2
Acesta este non-static b: 2
```

```
Acesta este static a: 2
Acesta este non-static b: 1
```

913 UTILIZAREA FUNCȚIILOR MEMBRE STATICE



După cum ați învățat, puteți declara date membre în cadrul claselor dumneavoastră ca *statice*. Puteți, de asemenea, să declarați funcții membre în cadrul claselor dumneavoastră ca *statice*. Compilatorul de C++ restricționează funcțiile pe care le declarați ca *statice* în câteva moduri:

1. Funcțiile statice pot accesa numai alți membri statici ai clasei.
2. Funcțiile membre statice nu au pointerul *this*.
3. Nu puteți supraîncărca o funcție statică cu o funcție non-statică sau invers.

Următorul program, *stat_fun.cpp*, este o versiune modificată a programului *stat_mem.cpp* care apare în secțiunea 912. Programul *stat_fun.cpp* declară funcția *arata* ca *statică*, astfel că programul poate accesa această funcție fie prin ea însăși, utilizând numai operatorul de rezoluție de clasă, fie în conexiune cu un singur obiect, ca mai jos:

```
#include <iostream.h>

class partajata
   {
     static int a;
     int b;
   public:
     void set(int i, int j)
     {
       a=i;
       b=j;
     }
     static void arata();
   };
int partajata::a;    //Defineste variabila globala
void partajata::arata()
   {
     cout << "Aceasta este static a: " << a << endl;
   }
void main(void)
   {
     partajata x, y;
     x.set(1,1);
     y.set(2,2);
     partajata::arata();
     y.arata();
     x.arata();
   }
```

DECLARAȚIILE FUNCȚIILOR MEMBRE

C/C++914

Așa cum ați învățat, o clasă conține variabile membre și funcții membre. Atunci când definiți funcțiile unei clase, puteți defini funcția în afara definiției clasei:

```
class film
  {
    public:
      char nume[64];
      char prim_actor[64];
      char aldoilea_actor[64];
      void arata_film(void);
      void initializare(char *nume, char *prim, char *aldoilea);
  };
void film::arata_film(void)
  {
    cout << "Numele filmului: " << nume << endl;
    cout << "Interpreteaza: " << prim_actor << " si " <<
          aldoilea_actor << endl << endl;
  }
void film::initializare(char *nume_film, char *prim,
  char *aldoilea)
  {
    strcpy(nume, nume_film);
    strcpy(prim_actor, prim);
    strcpy(aldoilea_actor, aldoilea);
  }
```

În acest caz, definiția clasei trebuie să conțină prototipul care descrie fiecare funcție membru a clasei. De asemenea, definiția funcției trebuie să specifice numele clasei înaintea numelui funcției.

UTILIZAREA DECLARAȚIILOR DE FUNCȚII INLINE

C/C++915

În secțiunea 914 ați învățat cum se definesc funcțiile unei clase în afara definiției clasei. Puteți să definiți funcțiile membre ale unei clase și în cadrul clasei, de fapt plasând instrucțiunile funcției în cadrul declarației clasei. De exemplu, următorul program, *inline.cpp*, definește funcțiile membru ale clasei *inline*, în cadrul declarației clasei:

```
#include <iostream.h>
#include <string.h>

class film
  {
    public:
      char nume[64];
      char primul_actor[64];
      char aldoilea_actor[64];
```

```
    void arata_film(void)
      {
        cout << "Numele filmului: " << nume << endl;
        cout << "Interpreteaza: " <<primul_actor << " si " <<
              aldoilea_actor << endl << endl;
      }
    void initializare(char * nume_film, char *primul,
      char *aldoilea)
      {
        strcpy(nume, nume_film);
        strcpy(primul_actor, primul);
        strcpy(aldoilea_actor, aldoilea);
      }
  };
void main(void)
  {
    film fugar, neobosit;
    fugar.initializare("Fugarul", "Harrison Ford", "Tommy Lee
      Jones");
    nopti.initializare("Nopti albe in Seattle", "Tom Hanks",
        "Meg Ryan");
    cout << "Ultimele doua filme pe care le-am vazut sunt: "
          << fugar.nume << " si " << nopti.nume << endl;
    cout << "Cred ca" << fugar.primul_actor << " a fost mare!"
          << endl;
  }
```

916 CÂND SE FOLOSESC FUNCȚIILE INLINE SAU EXTERIOARE



După cum ați învățat în secțiunea 915, atunci când declarați o funcție membru *inline*, instrucțiunile funcției se situează în cadrul clasei înseși. Un avantaj al declarării funcțiilor membre *inline* este acela că funcțiile *inline* ajută la concentrarea întregii funcții într-o singură locație în cadrul codului. Din păcate, utilizarea în acest mod a funcțiilor *inline* crește dimensiunea și complexitatea definiției claselor. Pe scurt, cu cât definițiile claselor dumneavoastră devin mai lungi, cu atât ele vor fi mai dificil de înțeles. În plus, la fel ca în cazul tipurilor obiectelor, codul nu este partajat de funcțiile *inline*.

Pe de altă parte, atunci când definiți o funcție membru în afara clasei, compilatorul de C++ creează o copie pentru fiecare instrucțiune a funcției. Fiecare obiect pe care îl creează ulterior programul pe baza acelei clase utilizează o singură copie a funcției. Cu alte cuvinte, când creați 1000 de obiecte, fiecare obiect partajează singura copie a codului funcției. O astfel de partajare a funcției este de preferat pentru că reduce semnificativ supraîncărcarea memoriei programului dumneavoastră.

CLASELE ȘI UNIUNILE

C/C++917

După cum ați învățat, structurile C++ sunt, în esență, clasele C++. În același fel, puteți folosi de asemenea uniuni C++ pentru a declara clase. Uniunile pot conține funcții constructor și destructor. Uniunile din C++ rețin toate caracteristicile de tip C (pe care le-ați învățat în secțiunile 481-487), inclusiv caracteristica de a menține toate elementele de date în aceeași locație de memorie. Membrii uniunilor, asemenea celor ai structurilor, dar spre deosebire de membrii claselor, sunt implicit publici. Așa cum ați învățat, una dintre cele mai bune utilizări ale uniunilor este operarea cu numere, folosind operații pe biți. Următorul program, *un_clas.cpp*, utilizează uniunea *schimb_octet* pentru a opera cu numere utilizând operații pe biți:

```
#include <iostream.h>

union schimb_octet
  {
    void schimb;
    void set_octet(unsigned i);
    void arata_cuvant(void);
    unsigned u;
    unsigned char c[2];
  };
void schimb_octet::schimb()
  {
    unsigned char t;
    t = c[0];
    c[0] = c[1];
    c[1] = t;
  }
void schimb_octet::arata_cuvant()
  {
    cout << u;
  }
void schimb_octet::set_octet(unsigned i)
  {
    u = i;
  }
void main(void)
  {
    schimb_octet b;
    b.set_octet(49034);
    b.schimb();
    b.arata_cuvant();
  }
```

PREZENTAREA UNIUNILOR ANONIME

C/C++918

C++ acceptă un tip special de uniuni numite *uniuni anonime*. O uniune anonimă nu conține un nume de tip, iar programul dumneavoastră nu poate declara variabile pe baza uniunii

anonime. În schimb, uniunea anonimă spune compilatorului că variabilele membre ale uniunii partajează aceeași locație. Programul se va referi însă în mod direct la variabile, fără obișnuita sintaxă cu operatorul punct. Pentru a înțelege uniunile anonime, analizați următorul program, *anon_un.cpp*, prezentat în continuare:

```
#include <iostream.h>
#include <string.h>

void main(void)
  {
  // defineste uniunea anonima
  union
    {
     long l;
     double d;
     char s[4];
    };
  // acum, programul se poate referi direct la elemente
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;
  }
```

919 PREZENTAREA FUNCȚIILOR FRIEND

C/C++

Este posibil să permiteți accesul la membrii privați ai unei clase, unei funcții care nu este membru. Pentru a face acest lucru în cadrul programelor dumneavoastră, puteți declara o clasă *friend* (prietenă). Funcțiile friend au acces la toți membrii privați și protejați ai clasei pentru care ele sunt declarate friend. Pentru a declara o funcție friend, includeți prototipul său în cadrul clasei, precedându-l de cuvântul cheie *friend*, cum arătăm în programul *frd_fun.cpp*:

```
#include <iostream.h>

class exemplu
  {
   int a, b;
   public:
    friend int suma(exemplu x);
    void set_ab(int i, int j);
  };
void exemplu::set_ab(int i, int j)
  {
   a = i;
   b = j;
  }
```

```
int suma(exemplu obiect)
   {
      /* Deoarece suma este un friend al clasei exemplu,
         ea poate accesa a si b direct */
      return obiect.a + obiect.b;
   }
void main(void)
   {
      exemplu intreg;

      cout << "Aduna 3 cu 4:" << endl;
      intreg.set_ab(3,4);
      cout << "Rezultat = " << suma(intreg);
   }
```

Când compilați și executați programul *frd_fun.cpp*, programul va afișa următoarea ieșire:

```
Aduna 3 cu 4:
Rezultat = 7
C:\>
```

Deși în cazul de mai sus nu există nici un motiv special pentru care facem funcția *suma* friend și nu funcție membru, există în general trei motive importante pentru utilizarea funcțiilor friend în cadrul claselor:

1. Funcțiile friend pot fi de folos atunci când programul supraîncarcă anumiți operatori, deoarece adăugarea unor funcții friend de control depășește acțiunile operatorului supraîncărcat.

2. Funcțiile friend fac mai ușoară crearea unor anumite funcții I/O.

3. Funcțiile friend pot să fie utile, de asemenea, în cazurile în care două sau mai multe clase conțin membri corelați cu alte părți ale programului; ele permit evitarea declarării mai multor funcții cu același cod.

În secțiunile ulterioare, veți învăța mai multe despre modul de utilizare a funcțiilor friend în fiecare dintre aceste cazuri.

PREZENTAREA CLASELOR FRIEND

C/C++920

După cum ați învățat, o clasă conține date și metode publice și private. De obicei, unica modalitate de accesare a membrilor privați este prin intermediul metodelor publice sau a interfețelor. Pe măsură ce programele dumneavoastră încep să lucreze cu mai multe tipuri de obiecte, se va ivi probabil situația în care un obiect apelează un alt obiect sau utilizează datele membre ale altui obiect. În secțiunile următoare, de exemplu, obiectul *Cititor* utilizează metoda *arata_carte* a obiectului *Carte* pentru a afișa titlul cărții. Unica modalitate în care obiectul *Cititor* poate accesa datele private ale obiectului *Carte* este prin intermediul metodei *arata_carte*. În funcție de programul dumneavoastră, poate că uneori veți dori ca un obiect să aibă acces la datele publice și private ale altui obiect. În astfel de cazuri, puteți specifica un *obiect friend (prieten)*. Dat fiind exemplul următor, cu *Cititor* și *Carte*, obiectul *Carte* poate declara obiectul *Cititor* ca friend. Obiectul *Cititor* poate după aceea să acceseze direct datele private ale obiectului *Carte*, afișând titlul cărții fără să mai apeleze metoda

arata_carte. Restul codului programului poate să nu aibă acces direct la datele private ale obiectului *Carte*. Unicul obiect care poate accesa datele private va fi prietenul obiectului *Carte*, obiectul *Cititor*. Însă, înainte de a specifica un prieten, ar trebui să informați compilatorul asupra clasei prietene, așa cum se explică în secțiunile următoare.

921 FUNCȚIILE CONSTRUCTOR

C/C++

Atunci când programele dumneavoastră creează o instanță obiect, programul va atribui apoi, de obicei, valorile inițiale la datele membre ale obiectului. Pentru a simplifica procesul de inițializare a membrilor obiectelor, C++ acceptă o funcție specială, numită constructor, care se execută automat ori de câte ori programul dumneavoastră creează o instanță de clasă. Funcția constructor este o metodă publică ce utilizează același nume cu clasa. De exemplu, utilizând clasa *Carte*, funcția constructor va avea același nume, *Carte*, ca mai jos:

```
class Carte
  {
   public:
     Carte(char *titlu, char *autor, char *editura, float pret);
             // Constructor
     char *titlu[256];
     char autor[64];
     float pret;
     void arata_titlu(void) { cout << titlu << '\n'; };
     float da_pret(void) { return(pret); };
     void arata_carte(void)
       {
         arata_titlu();
         arata_editura();
       };
     void atrib_editura(char *nume) { strcpy(editura, nume); };
   private:
     char editura[256];
     void arata_editura(void) { cout << editura << '\n'; };
  };
```

Programul poate defini funcția constructor în cadrul clasei înseși sau în afara ei. Atunci când programele declară ulterior un obiect, el poate transmite parametri către funcția constructor. Apoi, funcția constructor se va executa automat. Puteți transmite parametri către constructor, ca mai jos:

```
Carte capitole("Jamsa's C/C++ Programmer's Bible", "Jamsa si
          Klander", "Jamsa Press", 49.95);
```

Secțiunea 922 prezintă un program care utilizează o funcție constructor pentru a inițializa instanțe ale clasei *Carte*.

UTILIZAREA FUNCȚIILOR CONSTRUCTOR CU PARAMETRI

C/C++ 922

În secțiunea 921, ați învățat că un program poate să transmită parametri către funcția constructor. Este posibil, de asemenea, să transmiteți argumente funcției constructor. Programele vor utiliza aceste argumente în mod special pentru a ajuta la inițializarea unui obiect pe care ele îl creează. Pentru a crea *un constructor parametrizat*, adăugați pur și simplu parametri la declarația funcției constructor, cum faceți în cazul oricărei alte funcții. Atunci când definiți corpul funcției, utilizați parametrii pentru a inițializa obiectul. De exemplu, următoarea definiție a clasei inițializează clasa *Carte* în cadrul constructorului clasei:

```
class Carte
  {
    public:
      char *titlu[256];
      char autor[64];
      float pret;
      void Carte(char *titlu, char *autor, char *editura,
        float pret);
      void arata_titlu(void) { cout << titlu << '\n'; };
      float da_pret(void) { return(pret); };
      void arata_carte(void)
        {
          arata_titlu();
          arata_editura();
        };
      void atrib_editura(char *nume) { strcpy(editura, nume); };
    private:
      char editura[256];
      void arata_editura(void) { cout << editura << '\n'; };
  };
```

Dacă nu folosiți o funcție constructor parametrizată în cadrul clasei, puteți inițializa întotdeauna valorile în cadrul clasei, după ce programul construiește obiectul.

UTILIZAREA FUNCȚIEI CONSTRUCTOR

C/C++ 923

După cum ați învățat, o funcție constructor este o funcție specială a unei clase care se execută automat când creați o instanță a clasei respective. Programele utilizează în mod normal funcții constructor pentru a inițializa valorile membrilor. Următorul program, *construc.cpp*, utilizează funcția constructor *Carte* pentru a inițializa membri ai instanțelor clasei *Carte*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte {
  public:
    char titlu[256];
```

```
      char autor[64];
      float pret;
      Carte(char *btitlu, char *bautor, char *beditura, float bpret);
      void arata_titlu(void) { cout << titlu << '\n'; };
      float da_pret(void) { return(pret); };
      void arata_carte(void)
        {
          arata_titlu();
          arata_editura();
        };
      void atrib_editura(char *nume) { strcpy(editura, nume); };
    private:
      char editura[256];
      void arata_editura(void) { cout << editura << '\n'; };
  };
Carte::Carte(char *btitlu, char *bautor, char *beditura,
   float bpret)
   {
     strcpy(titlu, btitlu);
     strcpy(autor, bautor);
     strcpy(editura, beditura);
     pret = bpret;
   }
void main(void)
   {
     Carte capitole("Jamsa's C/C++ Programmer's Bible",
               "   Jamsa si Klander", "Jamsa Press", 49.95);
     Carte jurnal("All My Secrets...", "Kris Jamsa", "Fara", 9.95);
     capitole.arata_carte();
     jurnal.arata_carte();
   }
```

În programul *construc.cpp*, funcția constructor *Carte* precede fiecare nume al parametrilor săi cu litera *b* pentru a le distinge de numele membrilor clasei. Așa cum ați învățat, programul dumneavoastră va preceda numele variabilelor cu numele clasei pentru a rezolva conflictul de nume.

924 CÂND EXECUTĂ PROGRAMUL O FUNCȚIE CONSTRUCTOR

C/C++

Ca regulă generală, compilatorul va apela constructorul unui obiect atunci când codul programului declară obiectul. Însă, momentul real la care compilatorul apelează codul constructor poate să difere, în funcție de tipul clasei și de locația în cadrul proiectului. De exemplu, o funcție constructor a unui obiect local se execută când contorul programului întâlnește instrucțiunea de declarare a obiectului. În plus, dacă programul creează două sau mai multe obiecte în aceeași instrucțiune, programul va executa funcțiile constructor pentru fiecare obiect în ordinea declarării, de la stânga la dreapta.

Pentru obiectele globale, programul execută funcția constructor înainte ca funcția *main* să înceapă execuția. La fel ca în cazul obiectelor locale, constructorii globali se execută în ordine, de la stânga la dreapta și de sus în jos. Este imposibil de știut ordinea execuției pentru o serie de constructori globali răspândiți în mai multe fișiere cu cod sursă. Următorul program, *un_const.cpp*, prezintă execuția funcțiilor constructor:

```
#include <iostream.h>

class exemplu
  {
   public:
     int cine;
     exemplu(int id);
  } obiect_global1(1), obiect_global2(2);
exemplu::exemplu(int id)
  {
     cout << "Initializeaza " << id << "\n";
     cine = id;
  }
void main(void)
  {
     exemplu obiect_local(3);
     cout << "Aceasta NU este prima linie afisata.\n";
     exemplu obiect_local2(4);
  }
```

Observație: *Funcția constructor este o funcție specială a unei clase care se execută automat când programul creează o instanță. Funcțiile constructor nu returnează valori. Totuși, nu definiți funcții constructor ca returnând tipul* **void**. *În schimb, compilatorul de C++ poate determina că funcția este un constructor prin modul în care o folosiți. Prin definiție, nu puteți returna o valoare de la funcția constructor.*

UTILIZAREA FUNCȚIILOR CONSTRUCTOR CU PARAMETRI

C/C++ 925

După cum ați învățat în secțiunea 921, este posibil să transmiteți argumente funcției constructor. Programele vor utiliza aceste argumente în mod special pentru a ajuta la inițializarea unui obiect când ele îl creează. Pentru a crea un constructor parametrizat, adăugați parametri la declarația funcției constructor, cum faceți în cazul oricărei alte funcții. Atunci când definiți corpul funcției, utilizați parametrii pentru a inițializa obiectul. De exemplu, următoarea declarație a clasei inițializează obiecte ale clasei *Carte*:

```
Carte::Carte(char *btitlu, char *bautor, char *beditura,
  float bpret)
  {
    strcpy(titlu, btitlu);
    strcpy(autor, bautor);
    strcpy(editura, beditura);
```

```
    pret = bpret;
}
```

Dacă funcția constructor din cadrul exemplului anterior este numai o funcție constructor pe care o creați pentru obiectul *Carte*, programele dumneavoastră trebuie să declare fiecare instanță cu valori în cadrul declarației, care corespund cu parametrii *btitlu, bautor, beditura, bpret.* Dacă nu, compilatorul va returna o eroare.

926 REZOLVAREA CONFLICTELOR DE NUME ÎN FUNCȚIILE CONSTRUCTOR

C/C++

În secțiunile precedente, ați creat funcția constructor *Carte*, apoi ați modificat-o pentru a inițializa membrii pentru o instanță a clasei *Carte*. Pentru a face diferența între parametrii și numele membrilor clasei, programul a precedat numele fiecărui parametru cu litera *b*:

```
Carte::Carte(char *btitlu, char *bautor, char *beditura,
  float bpret)
  {
    strcpy(titlu, btitlu);
    strcpy(autor, bautor);
    strcpy(editura, beditura);
    pret = bpret;
  }
```

În cazul prezentat mai sus, numele parametrilor *titlu, autor, editura* și *pret* sunt mai semnificative și de aceea preferabile față de *btitlu, bautor, beditura* și *bpret*. Însă, pentru că numele parametrilor fără inițiala *b* s-ar confunda cu numele membrilor, funcția trebuie să rezolve aceasta utilizând numele clasei și două puncte duble, ca mai jos:

```
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
  {
    strcpy(Carte::titlu, titlu);
    strcpy(Carte::autor, autor);
    strcpy(Carte::editura, editura);
    Carte::pret = pret;
  }
```

CD-ROM-ul însoțitor al acestei cărți conține programul *constr2.cpp*, care prezintă codul sursă complet al programului pe care îl folosiți pentru a accesa obiectele de tip *Carte* (Book) atunci când utilizați funcția constructor pentru clasa *Carte*.

927 UTILIZAREA UNEI FUNCȚII CONSTRUCTOR PENTRU ALOCAREA MEMORIEI

C/C++

După cum ați învățat, funcțiile constructor permit programelor dumneavoastră să inițializeze variabile membre. Dacă variabila membru utilizează matrice, funcția constructor poate aloca volumul de memorie pe care îl doriți. De exemplu, următorul program, *cons_new.cpp*, utilizează operatorul *new* în cadrul funcției constructor *Carte* pentru a aloca memorie pentru o matrice de șiruri de caractere:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>
class Carte {
  public:
    char *titlu;
    char *autor;
    float pret;
    Carte(char *titlu, char *autor, char *editura, float pret);
    void arata_titlu(void) { cout << titlu << '\n'; };
    float da_pret(void) { return(pret); };
    void arata_carte(void)
      {
        arata_titlu();
        arata_editura();
      };
    void atrib_editura(char *nume) { strcpy(editura, nume); };
  private:
    char *editura;
    void arata_editura(void) { cout << editura << '\n'; };
};
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
  {
    if ((Carte::titlu = new char[256]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
    if ((Carte::autor = new char[64]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
    if ((Carte::editura = new char[128]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
    strcpy(Carte::titlu, titlu);
    strcpy(Carte::autor, autor);
    strcpy(Carte::editura, editura);
    Carte::pret = pret;
  }
void main(void)
  {
```

```
    Carte capitole("Jamsa's C/C++ Programmer's Bible",
              "Jamsa si Klander", "Jamsa Press", 49.95);
    Carte jurnal("All My Secrets...", "Kris Jamsa", "Fara", 9.95);
    capitole.arata_carte();
    jurnal.arata_carte();
  }
```

928 CONTROLUL CORECT AL ALOCĂRII MEMORIEI

În secțiunea 927, programul *cons_new.cpp* a utilizat operatorul *new* în cadrul unei funcții constructor pentru a aloca memorie pentru membrii șir de caractere. Codul pentru fiecare alocare de memorie a fost similar, ca mai jos:

```
    if ((Carte::titlu = new char[256]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
    if ((Carte::autor = new char[64]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
    if ((Carte::editura = new char[128]) == 0)
      {
        cerr << "Eroare la alocarea memoriei\n";
        exit(0);
      }
```

O modalitate de reducere a codurilor duplicate este de a încerca alocarea memoriei pentru fiecare variabilă și apoi testarea făcută după ultima alocare pentru a vedea dacă alocările au reușit, după cum urmează:

```
Carte::titlu = new char [256];
Carte::autor = new char [64];
Carte::editura = new char [128];
if((Carte::titlu && Carte::autor && Carte::editura) = = 0)
   {
     cout<<"Eroare de alocare a memoriei\n";
     exit(1);
   }
```

O a doua modalitate de a reduce codul este de a atribui mai întâi un handler propriu care va afișa mesajul de eroare și se va încheia atunci când spațiul liber poate satisface solicitările. Ați învățat cum se atribuie un handler propriu în secțiunea 871, unde s-a prezentat în detaliu funcția *set_new_handler*.

VALORILE IMPLICITE ALE PARAMETRILOR PENTRU CONSTRUCTORI

C/C++929

O funcție constructor este o metodă specială a unei clase, care se execută automat atunci când programul creează o instanță unui obiect. După cum ați învățat, C++ vă permite să dispuneți de valori implicite pentru parametrii funcțiilor. Funcțiile constructor nu fac excepție. Următorul program, *def_cons.cpp*, utilizează valorile implicite 1, 2 și 3 pentru membrii clasei *NumereMagice*:

```
#include <iostream.h>
#include <iomanip.h>

class NumereMagice {
  public:
    NumereMagice(int a = 1, int b = 2, int c = 3)
      {
        NumereMagice::a = a;
        NumereMagice::b = b;
        NumereMagice::c = c;
      };
    void arata_Numere(void)
      {
        cout << a << ' ' << b << ' '<< c << '\n';
      };
  private:
    int a, b, c;
};
void main(void)
  {
    NumereMagice unu(1, 1, 1);
    NumereMagice implicite;
    NumereMagice norocos(101, 101, 101);
    unu.arata_Numere();
    implicite.arata_Numere();
    norocos.arata_Numere();
  }
```

După cum puteți vedea, instanțele *unu* și *norocos* specifică valorile proprii ale membrilor. Instanța numită *implicite* utilizează, însă, valorile implicite 1, 2 și 3. Prin furnizarea în acest mod a valorilor implicite pentru funcțiile constructor, puteți să vă asigurați că programul dumneavoastră va inițializa întotdeauna membrii claselor la valori care au sens.

SUPRAÎNCĂRCAREA FUNCȚIILOR CONSTRUCTOR

C/C++930

După cum ați învățat, o funcție constructor este o metodă specială a claselor care se execută automat atunci când programul creează instanțe ale unui obiect. După cum ați învățat, C++ permite programelor dumneavoastră supraîncărcarea funcțiilor astfel încât compilatorul va decide ce funcție să invoce, în funcție de parametrii transmiși. Funcțiile constructor nu fac

excepție. Următorul program, *cereval.cpp*, dispune de două funcții constructor pentru clasa *Carte*. Prima funcție constructor atribuie valorile transmise ca parametri. A doua funcție constructor afișează un mesaj prin care se afirmă că programul trebuie să dispună de valori inițiale pentru fiecare parametru și apoi se încheie. În programul *cereval.cpp*, al doilea constructor se execută numai dacă programul încearcă să execute o funcție fără să specifice valori inițiale, ca mai jos:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>

class Carte {
  public:
    Carte(char *titlu, char *autor, char *editura, float pret);
    Carte(void);
    void arata_titlu(void) { cout << titlu << '\n'; };
    float da_pret(void) { return(pret); };
    void arata_carte(void)
      {
        arata_titlu();
        arata_editura();
      };
  private:
    char titlu[256];
    char autor[64];
    float pret;
    char editura[256];
    void arata_editura(void) { cout << editura << '\n'; };
};
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
  {
    strcpy(Carte::titlu, titlu);
    strcpy(Carte::autor, autor);
    strcpy(Carte::editura, editura);
    Carte::pret = pret;
  }
Carte::Carte(void)
  {
    cerr << "Trebuie sa specificati valori initiale pentru
         instanta Carte\n";
    exit(1);
  }
void main(void)
  {
    Carte capitole("Jamsa's C/C++ Programmer's Bible",
                   "Jamsa si Klander", "Jamsa Press", 49.95);
```

```
    Carte jurnal;
    capitole.arata_carte();
    jurnal.arata_carte();
  }
```

AFLAREA ADRESEI UNEI FUNCȚII SUPRAÎNCĂRCATE

C/C++ 931

După cum ați învățat, puteți atribui adresa unei funcții la un pointer și apoi să invocați acel pointer pentru a apela funcția. Însă, când supraîncărcați funcțiile în cadrul programelor dumneavoastră, obținerea adresei unei funcții este o problemă mai complexă. Pentru a înțelege mai bine de ce obținerea unei adrese este mai dificilă în contextul supraîncărcării funcțiilor, să ne oprim asupra următoarei instrucțiuni care atribuie adresa funcției *functie* la un pointer *p*:

```
p = functie;
```

Dacă funcția *functie* nu este supraîncărcată, instrucțiunea precedentă este suficientă pentru atribuirea adresei. Dacă, pe de altă parte, există mai multe funcții *functie*, compilatorul va refuza să compileze, pentru că nu poate rezolva referința. Răspunsul la această problemă constă în declararea variabilei înseși, așa cum prezentăm în următorul program, *over_pt.cpp*:

```
#include <iostream.h>

int functie(int a);
int functie(int a, int b);
void main(void)
  {
    int (*pointer_fisier)(int a); // pointer la int xxx(int)
    pointer_fisier = functie;     // indica functia(int)
    cout << pointer_fisier(5);
  }
int functie(int a)
  {
    return a;
  }
int functie(int a, int b)
  {
    return a*b;
  }
```

Deoarece programul declară *pointer_fisier* ca pointer la o funcție care returnează o valoare *int* și primește un singur parametru *int*, compilatorul poate rezolva pointerul în momentul în care o altă instrucțiune din program referențiază pointerul. Dacă, pe de altă parte, declararea lui *pointer_fisier* ar fi cea de mai jos, pointerul ar indica a doua funcție:

```
int(*pointer_fisier)(int a, int b);
```

932 UTILIZAREA FUNCȚIILOR CONSTRUCTOR CU UN SINGUR PARAMETRU

C/C++

În secțiunile precedente, ați învățat modul de parametrizare a funcțiilor constructor, pentru inițializarea datelor membre ale unei clase, de fiecare dată când programul creează o instanță a acelei clase. Însă, programele dumneavoastră pot, de asemenea, manipula funcțiile constructor cu numai un singur parametru, ca și cum declararea clasei ar fi o atribuire normală la un tip. Pentru a înțelege mai bine modul în care programele dumneavoastră vor implementa funcții constructor cu un singur parametru, să analizăm programul, *sing_par.cpp*, prezentat mai jos:

```
#include <iostream.h>
class simpla
  {
    int a;
    public:
      simpla(int j) {a = j;}
      int reda_a(void) {return a;}
  };
void main(void)
  {
    simpla ob = 99; // trece valoarea 99 in j
    cout << ob.reda_a();
  }
```

933 FUNCȚIILE DESTRUCTOR

C/C++

Așa cum ați învățat, de fiecare dată când creați o instanță obiect, programul dumneavoastră poate executa automat o funcție constructor pe care o puteți utiliza pentru inițializarea membrilor instanței. Într-un mod asemănător, C++ vă permite definirea unei funcții destructor care va rula automat atunci când programul distruge instanța. Funcțiile destructor rulează de obicei într-una din următoarele două situații: fie când programul se termină, fie când utilizați operatorul *delete* pentru a elibera memoria anterior alocată instanței. Funcțiile destructor au aceeași denumire ca și clasa. Diferențierea destructorilor față de constructori se face cu ajutorul caracterului *tilda* (~) care trebuie să preceadă numele fiecărei funcții destructor. De exemplu, următorul fragment de cod, prezintă declarațiile pentru funcțiile constructor și destructor:

```
Carte(char *titlu, char *autor, char *editura, float pret);
~Carte(void);
```

Așa cum puteți vedea, funcția destructor nu acceptă parametri și, la fel ca funcțiile constructor, declararea funcțiilor destructor se face fără o valoare de returnat. Veți învăța mai mult despre funcțiile destructor în următoarele secțiuni.

UTILIZAREA UNEI FUNCȚII DESTRUCTOR

C/C++934

În secțiunea 933 ați învățat despre corespondentul funcției constructor, și anume, funcția destructor. C++ apelează automat funcția destructor de fiecare dată când programul distruge instanța clasei. Pentru a înțelege mai bine această prelucrare, analizați următorul program, *destruct.cpp*. El creează o funcție destructor simplă, care afișează un mesaj care comunică faptul că programul distruge o instanță. Programul invocă automat funcția destructor pentru fiecare instanță, la terminarea programului, așa cum se prezintă mai jos:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte {
  public:
    char titlu[256];
    char autor[64];
    float pret;
    Carte(char *titlu, char *autor, char *editura, float pret);
    ~Carte(void);
    void arata_titlu(void) { cout << titlu << '\n'; };
    float da_pret(void) { return(pret); };
    void arata_carte(void)
      {
        arata_titlu();
        arata_editura();
      };
    void atrib_editura(char *nume) { strcpy(editura, nume); };
  private:
    char editura[256];
    void arata_editura(void) { cout << editura << '\n'; };
};
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
  {
    strcpy(Carte::titlu, titlu);
    strcpy(Carte::autor, autor);
    strcpy(Carte::editura, editura);
    Carte::pret = pret;
  }
Carte::~Carte(void)
  {
    cout << "Distruge instanta " << titlu << '\n';
  }
void main(void)
  {
    Carte capitole("Jamsa's C/C++ Programmer's Bible",
        "Jamsa si Klander", "Jamsa Press", 49.95);
    Carte jurnal("All My Secrets...", "Kris Jamsa", "Fara", 9.95);
```

```
    capitole.arata_carte();
    jurnal.arata_carte();
}
```

935 NECESITATEA FUNCȚIILOR DESTRUCTOR

C/C++

În secțiunea 933 și 934 ați învățat despre funcțiile destructor. Așa cum ați învățat, C++ apelează automat funcțiile destructor de fiecare dată când renunță la instanța unei clase, tot așa cum apelează automat funcțiile constructor de fiecare dată când creează o instanță a unei clase. În cele mai multe cazuri, veți observa că funcția destructor nu execută o prelucrare specială. Dar, pe măsură ce programele dumneavoastră devin mai complexe, veți găsi două situații în care o clasă trebuie să aibă o funcție destructor.

Ca regulă generală, funcțiile destructor sunt mult mai importante pentru structura programului atunci când clasele alocă memorie dinamică. Când clasele dumneavoastră creează toate datele membre, matricele, structurile și celelalte la crearea fiecărei instanțe, C++ se va ocupa automat de o mare parte a procesului de distrugere. Dacă, însă, obiectele dumneavoastră alocă memorie pe măsura rulării (de exemplu, un obiect listă înlănțuită), ar trebui să vă asigurați că programul eliberează acea memorie (un proces pe care programatorii îl denumesc adesea *colectarea gunoiului*).

În plus, dacă programul utilizează o serie de obiecte înlănțuite, funcția destructor vă va ajuta să mențineți lista după distrugerea unui obiect din listă. După cum ați învățat, într-o listă simplu înlănțuită, fiecare obiect păstrează adresa obiectului care îl urmează în cadrul listei. În lista dublu înlănțuită, fiecare obiect păstrează atât adresa obiectului care urmează, cât și a celui care îl precede în cadrul listei. De fiecare dată când ștergeți un element (sau nod) dintr-o listă înlănțuită, programul dumneavoastră trebuie să actualizeze legăturile din cadrul listei pentru a evita ruperea listei. Așa cum veți învăța în secțiunile următoare, puteți efectua multe dintre operațiile specifice listelor înlănțuite în cadrul funcțiilor destructor.

936 CÂND INVOCĂ UN PROGRAM O FUNCȚIE DESTRUCTOR

C/C++

Atunci când creați funcții destructor pentru programele dumneavoastră, ar trebui să înțelegeți momentul la care programele vor invoca funcțiile destructor pe care le creați pentru clasele dumneavoastră. Pe scurt, C++ invocă funcțiile destructor ale obiectelor chiar înainte de renunțarea la respectivele obiecte. Pentru a înțelege mai bine modul de funcționare a funcțiilor destructor, analizați figura 936, o schiță logică simplă a ciclului de viață al unui obiect.

De aceea, în cadrul programelor dumneavoastră, ar trebui să vă asigurați atât de faptul că funcțiile destructor execută numai activitățile adecvate asupra obiectului ce urmează a fi distrus, cât și de faptul că programele dumneavoastră nu își propun execuția funcției destructor înainte de încheierea ciclului de viață a obiectului respectiv. Pentru a înțelege mai bine conceptul de ciclu de viață al obiectului, analizați următorul program, *stiva_cd.cpp*, care construiește și apoi distruge câteva obiecte de tip *stiva*:

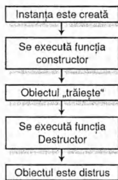


Figura 936 *Modelul logic al vieții unui obiect.*

```
#include <iostream.h>
#define DIM_TABLOU 100
class stiva {
    int stv[DIM_TABLOU];
    int stiva_top;
public:
    stiva();
    ~stiva();
    void depune(int i);
    int extrage();
};
stiva::stiva(void)
  {
    stiva_top = 0;
    cout << "Stiva initializata" << endl;
  }
stiva::~stiva(void)
  {
    cout << "Stiva distrusa" << endl;
  }
void stiva::depune(int i)
  {
    if (stiva_top== DIM_TABLOU)
      {
        cout << "Stiva este plina." << endl;
        return;
      }
    stv[stiva_top] = i;
    stiva_top++;
  }
int stiva::extrage(void)
```

```
   {
     if (stiva_top==0)
       {
         cout << "Stiva depasita." << endl;
         return 0;
       }
     stiva_top--;
     return stv[stiva_top];
   }
void main(void)
   {
     stiva ob1, ob2;
     ob1.depune(1);
     ob2.depune(2);
     ob1.depune(3);
     ob2.depune(4);
     cout << ob1.extrage() << endl;
     cout << ob1.extrage() << endl;
     cout << ob2.extrage() << endl;
     cout << ob2.extrage() << endl;
   }
```

Când compilați și executați programul *stiva_cd.cpp*, veți observa că programul va apela automat funcția destructor pentru două obiecte stivă, chiar înainte ca programul să își încheie execuția. Atunci când lucrați cu matrice de obiecte clasă, amintiți-vă că programul va activa funcția destructor pentru fiecare element din matricea de clase pe care programul o distruge. Cu alte cuvinte, dacă aveți o matrice cu 100 de elemente ale unei clase, atunci când programul va distruge matricea, el va apela funcția destructor de 100 de ori – câte o dată pentru fiecare element. Veți învăța mai multe despre lucrul cu matricele de clase în secțiunile următoare.

937 UTILIZAREA UNEI COPII A CONSTRUCTORULUI C/C++

În mod implicit, atunci când C++ face o copie unui obiect, el efectuează o *copie la nivel de biți*, ceea ce înseamnă că noul obiect este copia exactă a obiectului originar. În anumite cazuri, însă, copierea pe biți poate cauza mai multe probleme decât să o rezolve. De exemplu, dacă o funcție primește o instanță obiect prin valoare, apoi face o copie locală a instanței în interiorul funcției, când programul încheie funcția, el va șterge copia locală a obiectului așa cum v-ați aștepta. Însă, atunci când C++ șterge copia locală, el șterge și memoria utilizată de copia exterioară. Puteți preveni problemele de acest gen prin scrierea unei *funcții constructor copie*. Forma generală a unei funcții constructor copie este prezentată mai jos:

```
numeclasa (const numeclasa &obiect)
   {
     // corpul functiei constructor
   }
```

În acest caz, parametrul *obiect* este instanța obiect pe care vreți să o copiați. Puteți include și parametri de inițializare în cadrul constructorului copie, deși va trebui să furnizați valori implicite pentru fiecare parametru. Următorul program, *con_cop.cpp*, utilizează un constructor copie cu o matrice de clase:

```
#include <iostream.h>
#include <stdlib.h>

class matrice {
    int *p;
    int dimens;
  public:
    matrice(int dim) { // constructor simplu
      p = new int[dim];
      if(!p) exit(1);
      dimens = dim;
    }
    ~matrice() {delete [] p;} // destructor
    matrice(const matrice &obiect);// constructor copie
    void atrib(int i, int j){
      if(i>=0 && i<dimens)
        p[i] = j;
    }
    int reda(int i) {return p[i];}
};
matrice::matrice(const matrice &obiect)
   {
    int lcl_i;
    p = new int[obiect.dimens];
    if (!p)
      exit(1);
    for(lcl_i=0; lcl_i < obiect.dimens; lcl_i++)
      p[lcl_i] = obiect.p[lcl_i];
   }
void main(void)
   {
    matrice num(10);
    int lcl_i;
    for (lcl_i=0; lcl_i<10; lcl_i++)
      num.atrib(lcl_i, lcl_i);
    for (lcl_i=9; lcl_i>=0; lcl_i--)
      cout << num.reda(lcl_i);
    cout << endl;
    // Creeaza alta matrice utilizand constructorul copie
    tablou x=num;
    for (lcl_i=0; lcl_i<10; lcl_i++)
      cout << x.reda(lcl_i);
   }
```

Atunci când executați programul *con_cop.cpp*, el va crea mai întâi obiectul *num*, pe care îl inițializează și îl afișează. După aceea, programul utilizează *num* pentru a inițializa *x*, apelând constructorul copie în acest proces. Constructorul copie copiază toate datele din cadrul obiectului *num* în obiectul *x*, dar făcând aceasta, el creează un spațiu propriu de memorie pentru *x*, independent de cel al obiectului *num*.

Observație: *Programele dumneavoastră pot să apeleze un constructor copie numai pe parcursul inițializării. Dacă programul creează un obiect și apoi încercă să îi facă o copie într-un alt obiect, constructorul copie nu va interveni.*

938 UTILIZAREA CONSTRUCTORILOR DE TIP EXPLICIT

C/C++

Puteți utiliza *constructori de tip explicit* în cadrul programelor dumneavoastră pentru a forța toate declarațiile într-o formă stabilită de constructorul dumneavoastră. De obicei, atunci când creați un constructor, așa cum prezintă fragmentul de cod de mai jos, compilatorul permite mai multe stiluri de inițializare:

```
exempluclasa(in j) (I=j;)
//
// Instructiuni de program
exempluclasa ob1(10);
exempluclasa ob2 = 10;
//
// Alte instructiuni de program
```

Totuși, dacă declarați clasa ca explicită, compilatorul va permite programului să utilizeze constructori numai de tipul și formatul stabilit. Utilizarea claselor explicite este probabil alegerea cea mai fericită în bibliotecile de clase și în alte locații de clasă semi-fixate. Veți utiliza cuvântul cheie *explicit* pentru a defini o clasă:

```
explicit exempluclasa(int j) (I=j;)
```

939 DOMENIUL DE VALABILITATE AL UNEI CLASE

C/C++

După cum ați învățat, *domeniul de valabilitate* al unui identificator definește locația în cadrul programului pentru care identificatorul este recunoscut. Clasele C++, la fel ca și tipurile și variabilele, au un domeniu de valabilitate care începe la definirea lor din cadrul fișierului program și există până la sfârșitul blocului în care clasa a fost definită. Pentru a mări domeniul unei clase, puteți defini clasa în afara tuturor blocurilor programului. În plus, dacă definiți clasa de tip *extern*, clasa este recunoscută pe parcursul întregului program. Dacă definiți clasa de tip *static*, domeniul clasei rămâne același ca și când clasa ar fi *automatică*, dar ea va exista pe durata întregului program.

940 CLASELE IMBRICATE

C/C++

După cum ați învățat în capitolele anterioare ale acestei cărți, puteți defini o structură în cadrul alteia. În același fel, este posibil să definiți și o clasă în cadrul alteia. Definind o clasă în cadrul altei clase, creați o *clasă imbricată*. Deoarece declarația clasei definește, de fapt, domeniul ei, o

clasă imbricată este validă numai în interiorul domeniului clasei care o include. Din acest motiv, ar trebui să utilizați rareori clasele imbricate în programele dumneavoastră. Deoarece puteți utiliza flexibilitatea specifică limbajului C++, în special mecanismele sale de moștenire (despre care veți învăța în secțiunile următoare), nu este necesar să utilizați clasele imbricate.

CLASELE LOCALE

C/C++ 941

Așa cum puteți defini variabile locale în interiorul unei funcții, la fel puteți defini o clasă în cadrul unei funcții. Atunci când declarați o clasă în cadrul unei funcții, clasa este recunoscută numai în cadrul respectivei funcții. Următorul program, *cl_local.cpp*, definește o clasă locală validă:

```
#include <iostream.h>

void f(void);
void main(void)
  {
    f();
  }
void f(void)
  {
    class clasalocala
      {
        int i;
       public:
        void atrib_i(int n) {i=n;}
        int reda_i() {return i;}
      } ob;
    ob.atrib_i(10);
    cout << ob.reda_i();
  }
```

C++ aplică niște restricții claselor locale ceea ce le face să fie mai puțin utilizate în cadrul programelor C++:

1. Trebuie să definiți toate funcțiile membre în cadrul declarației clasei (cu alte cuvinte, toate funcțiile membre trebuie definite *inline*).
2. Clasele locale nu pot să utilizeze sau să acceseze variabile locale ale funcției în care este declarată.
3. Nu puteți să declarați nici o variabilă de tip *static* în clasa locală.

REZOLVAREA CONFLICTELOR DE NUME ALE MEMBRILOR ȘI PARAMETRILOR

C/C++ 942

În cadrul funcțiilor membre, este posibil ca uneori să existe conflicte între numele membrilor clasei și numele parametrilor transmiși funcției. În mod implicit, C++ rezolvă astfel de conflicte de nume utilizând parametrul (variabila locală) și ascunzând existența membrului clasei. Pentru a preveni astfel de conflicte de nume, precedați referința la membrul clasei cu numele clasei și două puncte duble, ca mai jos:

```
void pisici::atrib_pisici(char *rasa, int inaltime, int greutate)
   {
     strcpy(pisici::rasa, rasa);
     pisici::inaltime = inaltime;
     pisici::greutate = greutate;
   }
```

În acest caz, numele care este precedat de *pisici::* corespunde numelor membrilor clasei. Celelalte nume corespund variabilelor locale.

943 CREAREA UNEI MATRICE DE VARIABILE CLASĂ C/C++

Mai multe dintre secțiunile prezentate în această carte au creat matrice de structuri. În mod similar, programele dumneavoastră pot crea o matrice a instanțelor de clase. Următorul program, *biblio.cpp*, creează o matrice care conține specificații pentru patru cărți:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte
   {
     public:
       void arata_titlu(void)
         { cout << titlu << '\n'; };
       void arata_carte(void)
         { arata_titlu(); arata_editura(); };
       void atrib_membri(char *, char *, char *, float);
     private:
       char titlu[256];
       char autor[64];
       float pret;
       char editura[256];
       void arata_editura(void) { cout << editura << '\n'; };
};
void Carte::atrib_membri(char *titlu, char *autor, char *editura,
     float pret)
   {
     strcpy(Carte::titlu, titlu);
     strcpy(Carte::autor, autor);
     strcpy(Carte::editura, editura);
     Carte::pret = pret;
   }
void main(void)
   {
     Carte Biblioteca[4];
     Biblioteca[0].atrib_membri("Jamsa's C/C++ Programmer's Bible",
          "Jamsa si Klander", "Jamsa Press", 49.95);
     Biblioteca[1].atrib_membri("Hacker Proof", "Klander", "Jamsa
```

```
                            Press", 54.95);
    Biblioteca[2].atrib_membri("ActiveX Programmer's Library",
       "Lalani si Chandak", "Jamsa Press", 49.95);
    Biblioteca[3].atrib_membri("Rescued by C++",
       "Jamsa", "Jamsa Press", 24.95);
    for (int i = 0; i < 4; i++)
      Biblioteca[i].arata_carte();
  }
```

CONSTRUCTORII ȘI MATRICE DE CLASE

C/C++ 944

După cum ați învățat, C++ permite programelor dumneavoastră să declare matrice de un anumit tip de clasă. Atunci când declarați o matrice, C++ invocă automat funcția constructor pentru fiecare intrare a matricei. De exemplu, următorul program, *tabclas.cpp*, creează o matrice de tipul clasei *Angajat*:

```
#include <iostream.h>

class Angajat
  {
    public:
      Angajat(void) { cout << "Construieste o instanta\n"; };
      void arata_angajat(void) { cout << nume; };
    private:
      char nume[256];
      long id;
  };
void main(void)
  {
    Angajat muncitor[5];
    // Alte instructiuni
  }
```

Când compilați și executați programul *tabclas.cpp*, veți observa că programul va apela automat funcția constructor de cinci ori, câte o dată pentru fiecare element al matricei.

SUPRAÎNCĂRCAREA UNUI OPERATOR

C/C++ 945

După cum ați învățat, atunci când supraîncărcați o funcție, compilatorul de C++ stabilește ce funcție să invoce, bazându-se pe numărul parametrilor și tipul lor. Atunci când creați o clasă, C++ vă permite să supraîncărcați, de asemenea, operatorii. Când supraîncărcați un operator, trebuie să continuați să utilizați operatorul în formatul său standard. De exemplu, dacă supraîncărcați operatorul plus (+), acesta trebuie să utilizeze operatorul sub forma *operand+operand*. În plus, puteți să supraîncărcați numai operatorii existenți. C++ nu vă permite definirea unor operatori proprii. Operatorul supraîncărcat creat se aplică numai instanțelor clasei specificate. De exemplu, să presupunem că ați creat clasa *Sir* și ați supraîncărcat operatorul plus astfel încât operatorul să concateneze două șiruri de caractere:

```
sir_nou = sir + tinta;
```

Dacă utilizați operatorul supraîncărcat plus cu două valori întregi sau în virgulă mobilă, supraîncărcarea nu se va aplica. În plus, C++ nu va permite supraîncărcarea operatorilor listați în tabelul 945.

Operator	Funcție
.	Operator de membru al clasei
.*	Operator pointer la membru
::	Operator de rezoluție a domeniului de valabilitate
?:	Operator expresie condițională

Tabelul 945 *Operatorii pentru care C++ nu permite supraîncărcarea.*

946 CREAREA UNEI FUNCȚII OPERATOR MEMBRĂ

Atunci când creați funcții *operator* membre pentru a supraîncărca funcționarea unui operator, declarațiile membrilor operatori vor avea forma generală prezentată mai jos:

```
tip-return nume-clasa::operator #(lista-argumente)
   {
     // Operatii
   }
```

Adesea, funcțiile *operator* returnează un obiect al clasei asupra căruia operează. Însă, C++ vă permite definirea lui *tip-return* ca orice tip valid. Simbolul # reprezintă rezervarea unui spațiu pentru operatorul pe care doriți să îl supraîncărcați. De exemplu, în secțiunea 947, veți supraîncărca operatorul plus utilizând o declarație de funcție similară cu cea prezentată în continuare:

```
char *operator +(char *sir_ad)
```

În această declarație de funcție, membrul operator este semnul plus. Atunci când supraîncărcați un operator unar (deci, un operator care acționează numai asupra unei singure valori), *lista-argumente* trebuie să fie goală. Atunci când supraîncărcați un operator binar, *lista-argumente* trebuie să conțină numai un singur parametru.

Motivul acestei aparent ciudate construcții este acela că C++ trece automat valoarea din stânga operatorului către funcția supraîncărcată. De aceea, atunci când invocați un operator unar, C++ va transmite automat valoarea asupra căreia funcția operează către funcția supraîncărcată. Așa cum veți învăța în continuare, înțelegerea faptului că C++ transmite valorile către funcțiile supraîncărcate este în mod special important atunci când manipulați operatorii de incrementare și decrementare prefix și postfix.

947 SUPRAÎNCĂRCAREA OPERATORULUI PLUS

Ați învățat că pentru a supraîncărca un operator, trebuie să creați o clasă la care doriți ca supraîncărcarea să se aplice. După ce ați creat clasa, trebuie să plasați în cadrul metodelor ei publice o linie antet care să definească operatorul. De exemplu, următorul program, *spr_pls.cpp*, creează o clasă *Sir* și supraîncarcă operatorul plus (+) astfel că acesta concatenează șirurile:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
class Sir
  {
   public:
     Sir *operator +(char *sir_adaug)
     Sir(char *in_sir)
       { strcpy(buffer, in_sir);
         lung = strlen(buffer); }
     Sir(void) {lung = 0;};
     void arata_sir() { cout << buffer; };
   private:
     char buffer[256];
     int lung;
  };
Sir Sir::operator+(char *sir_adaug)
  {
   Sir temp;
   int lungtemp;
   lungtemp = strlen(buffer) + strlen(sir_adaug) + 1;
   if(lungtemp>256)
     {
       cout << "Sir prea lung!" << endl;
       strcpy(temp.buffer, buffer);
       return temp;
     }
   lung = lungtemp;
   strcpy(temp.buffer, buffer);
   strcat(temp.buffer, sir_adaug);
   return temp;
  }
void main(void)
  {
   Sir titlu("Jamsa's C/C++ ");
   titlu = titlu + "Programmer's Bible\n";
   titlu.arata_sir();
  }
```

Atunci când rulați programul *spr_pls.cpp*, el va începe prin atribuirea membrului *buffer* șirului „Jamsa's C/C++". Programul va utiliza apoi operatorul plus supraîncărcat pentru a concatena caracterele „Programmer's Bible". Observați că operatorul supraîncărcat este o funcție simplă care primește un parametru. Funcția primește numai un singur parametru. Parametrul este al doilea operand. Operația însăși implică operandul instanței.

Operatorul supraîncărcat *plus* utilizează funcțiile *strcpy* și *strcat* pentru a copia șirul de caractere dintre ghilimele în obiectul *titlu*. Observați că acest cod din cadrul funcției operator

plus supraîncărcate se referă la datele membre ale obiectului *titlu* în mod implicit, cu comenzi cum sunt următoarele, care plasează valoarea curentă a titlului în obiectul *temp*:

```
strcpy(temp.buffer, buffer);
```

Programul ar putea la fel de uşor să facă referire la obiect în mod explicit, utilizând pointerul *this*, ca mai jos:

```
strcpy(temp.buffer, this.buffer);
```

948 SUPRAÎNCĂRCAREA OPERATORULUI SEMN MINUS

C/C++

În secţiunea 947 aţi creat o clasă *Sir* şi aţi supraîncărcat operatorul plus. Următorul program, *sprminus.cpp*, supraîncarcă operatorul minus (–), apoi utilizează operatorul supraîncărcat pentru a elimina toate apariţiile unui caracter specificat din membrul *buffer* al clasei:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Sir
  {
    public:
      Sir operator +(char *adaug_sir)
      Sir operator -(char subsir);
      Sir(char *in_sir)
        { strcpy(buffer, in_sir);
          lungime = strlen(buffer); }
      Sir() {lungime = 0;};
      void arata_sir(void) { cout << buffer; };
    private:
      char buffer[256];
      int lungime;
  };
Sir Sir::operator + (char *adaug_sir)
  {
    Sir temp;
    int lung.temp;
    lung temp = strlen(buffer) + strlen(adaug_sir) + 1;
    if (lungtemp > 256)
      {
        count << "Sir prea lung!" << endl;
        strcpy(temp.buffer, buffer);
        lungime = strlen(buffer);
        return temp;
      }
    lungime = lungtemp;
    strcpy(temp.buffer, buffer);
```

```
    strcat(temp.buffer, adaug_sir);
    return temp;
  }
Sir Sir::operator -(char subsir)
  {
    Sir temp
    char *S1;
    int i, j;
    s1 = buffer;
    for (i = 0, *s1 ; i++)
      {
        if (*s1 ! = *subsir)
          {
            temp.buffer[i]=*s1;
            s1++;
          }
        else
          {
            for (j = 0; subsir[j] = = S1[j]&& subsir [j]; j++)
              ;
            if (! subsir [j])
              {
                s1 + = j;
                i --;
              }
            else
              {
                temp.buffer [i] = *s1;
                s1 ++ ; }
              }
          }
      }
      temp.buffer[i] = '\0';
      temp.lungime = strlen(temp.buffer);
      return temp;
  }
void main(void)
  {
    Sir titlu("Jamsa's C/C++ ");
    titlu = titlu + "Programmer's Bible\n";
    titlu.arata_sir();
    titlu = titlu - 's';
    titlu.arata_sir();
  }
```

Atunci când rulați programul *sprminus.cpp*, el va începe prin a atribui membrului *buffer* șirul Jamsa's C/C++". Programul va utiliza apoi operatorul plus supraîncărcat pentru a concatena caracterele „Programmer's Bible". Observați că operatorul supraîncărcat este o funcție

simplă care primește un parametru. Funcția primește numai un singur parametru. Parametrul este al doilea operand. Operația însăși implică operandul instanță.

Operatorul supraîncărcat minus utilizează o buclă simplă *for* pentru a se deplasa într-o matrice de tip *char*, element cu element. Dacă elementul nu corespunde cu prima literă din subșir, el copiază elementul într-un obiect *temp*, apoi se deplasează spre următorul element. Dacă elementul corespunde cu prima literă din subșir, programul intră într-o a doua buclă care compară elementele cu elementele din subșir. Dacă ele corespund, procesul de copiere sare peste întregul subșir. Dacă nu corespund, procesul de copiere se returna la primul element și începe prelucrarea de la această poziție din matricea de tip *char*. Observați că acest cod din funcția de supraîncărcare a operatorului minus se referă implicit la datele membre ale obiectului *titlu*, cu comenzi care plasează valoarea curentă a obiectului *titlu* în obiectul *temp*:

```
s1 = buffer;
```

Programul ar putea la fel de ușor să se refere explicit la obiect, utilizând pointerul *this*, ca în exemplul de mai jos:

```
s1 = this.buffer;
```

949 SUPRAÎNCĂRCAREA OPERATORILOR DE INCREMENTARE PREFIX ȘI POSTFIX

C/C++

După cum ați învățat, este posibil să supraîncărcați operatori și funcții în C++. Una dintre cele mai frecvent utilizate perechi de operatori în C++ este cea a operatorilor de incrementare prefix și postfix. Așa cum ați învățat, dacă plasați operatorul de incrementare (++) înaintea unei variabile, C++ incrementează variabila înainte de a o interpreta; dacă plasați operatorul după variabilă, C++ interpretează variabila înaintea de a o incrementa.

Versiunile mai vechi de C++ nu ofereau programatorilor mijloace de supraîncărcare diferite pentru operatorii de incrementare prefix și postfix. Versiunile moderne de C++, însă, vă pun la dispoziție mijloace de determinare a poziționării operatorilor de incrementare, înaintea (prefix) sau după (postfix) operandul lor. Pentru a supraîncărca o incrementare prefix sau postfix, veți defini două versiuni de funcție *operator++*, ca mai jos:

```
nume-clasa operator++();
nume-clasa operator++(int x);
```

Dacă operatorul de incrementare precede operandul, compilatorul va apela funcția *operator++()*. Dacă, însă, operatorul de incrementare urmează operandului, compilatorul va apela funcția *operator++(int x)*. Următorul program, *suprainc.cpp*, supraîncarcă operatorul de incrementare pentru clasa *sir*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Sir
  {
    public:
```

```
    Sir Sir::operator++()
      { strcat(buffer, "X");
        return *this; };
    Sir Sir::operator++(int x)
      { strcat(buffer, "X");
        return *this; };
    Sir(char *sir)
      { strcpy(buffer, sir);
        lung = strlen(buffer); }
    void arata_sir(void) { cout << buffer << endl; };
  private:
    char buffer[256];
    int lung;
  };
void main(void)
  {
   Sir titlu("Jamsa's C/C++ Programmer's Bible");
   titlu++;
   titlu.arata_sir();
   ++titlu;
   titlu.arata_sir();
  }
```

SUPRAÎNCĂRCAREA OPERATORILOR DE DECREMENTARE PREFIX ȘI POSTFIX

C/C++950

După cum ați învățat în secțiunea 949, versiunile moderne de C++ vă pun la dispoziție o modalitate de a supraîncărca atât tipul prefix, cât și tipul postfix al unui operator dat. La fel cum declarați două funcții operator pentru supraîncărcarea operatorului de incrementare, la fel veți crea două funcții operator pentru supraîncărcarea operatorului de decrementare:

```
nume-clasa operator--();
nume-clasa operator--(int x);
```

Dacă operatorul de decrementare precede operandul, compilatorul va apela funcția *operator--()*. Dacă, însă, operatorul de incrementare urmează operandului, compilatorul va apela funcția *operator--(int x)*. Următorul program, *supradec.cpp*, supraîncarcă operatorul de decrementare:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Sir
  {
  public:
    Sir Sir::operator--()
      { buffer[lung-1] = NULL;
```

```
      lung--;
      return *this; };
    Sir Sir::operator--(int x)
      { buffer[lung-1] = NULL;
      lung--;
      return *this; };
    Sir(char *sir)
      { strcpy(buffer, sir);
      lung = strlen(buffer); }
    void arata_sir(void) { cout << buffer << endl; };
  private:
    char buffer[256];
    int lung;
};
void main(void)
  {
    Sir titlu("Jamsa's C/C++ Programmer's Bible");
    titlu--;
    titlu.arata_sir();
    --titlu;
    titlu.arata_sir();
  }
```

951 SĂ RECAPITULĂM RESTRICȚIILE LA SUPRAÎNCĂRCAREA UNUI OPERATOR

C/C++

După cum ați învățat, C++ impune o limită operatorilor pe care programele dumneavoastră pot să îi supraîncarce. Așa cum ați învățat, nu puteți supraîncărca operatorul *punct*, operatorul *de rezoluție a domeniului de valabilitate*, operatorul *condițional* sau operatorul *pointer de redirectare*. Cu excepția acestor operatori, puteți să supraîncărcați orice operator doriți.

De exemplu, veți putea să supraîncărcați operatorul plus în așa fel încât să scrie pe ecran de zece ori „Happy este dalmatian". Însă, ar trebui, în general, să nu supraîncărcați un operator într-un mod fundamental diferit de utilizarea sa normală. Atunci când un alt programator citește codul dumneavoastră și vede *a+b*, ar fi normal să aștepte ca operatorul plus supraîncărcat să efectueze un tip de adunare – nu o serie de ieșiri pe ecran.

Cu excepția operatorului de atribuire, clasele derivate vor moșteni toți operatorii supraîncărcați de la clasa de bază. Însă, clasele derivate pot să supraîncarce ele însele orice operator (chiar operatori care au fost supraîncărcați de clasa bază).

952 UTILIZAREA FUNCȚIILOR FRIEND PENTRU SUPRAÎNCĂRCAREA OPERATORILOR

C/C++

Așa cum au arătat secțiunile precedente, puteți utiliza funcțiile *friend* (*prietene*) pentru supraîncărcarea operatorilor în cadrul claselor dumneavoastră. Este important, totuși, să înțelegem că există unele diferențe între supraîncărcarea unui operator în mod normal și supraîncărcarea cu o funcție *friend*. Cea mai importantă diferență este aceea că funcțiile

friend nu au acces la pointerul *this* pentru clasă. De aceea, programul trebuie să transmită explicit operanzii către funcția *friend* de supraîncărcare a operatorului. Cu alte cuvinte, o funcție *friend* care supraîncarcă un operator *unar* primește un parametru, iar un prieten care supraîncarcă un operator *binar* primește doi parametri. După cum ați învățat, un operator supraîncărcat în cadrul unei clase primește un parametru mai puțin decât așteaptă operatorul, datorită utilizării pointerului *this*. Atunci când programele dumneavoastră supraîncarcă un operator *binar* utilizând o funcție *friend*, programele trebuie să transmită operandul stâng în primul parametru și operandul drept în al doilea parametru. Următorul program, *frn_plus.cpp*, utilizează o funcție *friend* pentru a supraîncărca operatorul +:

```
#include <iostream.h>
class loc {
    int longitudine, latitudine;
  public:
    loc(void) {} // Utilizat pentru a construi temporare
    loc (int lg, int lt)
      {
        longitudine = lg;
        latitudine = lt;
      }
    void arata(void)
      {
        cout << longitudine << " ";
        cout << latitudine << endl;
      }
    friend loc operator+(loc op1, loc op2); // Friend de
                                           // supraincarcare
    loc operator=(loc op2);
  };
loc operator+(loc op1, loc op2)
  {
    loc temp;
    temp.longitudine = op1.longitudine + op2.longitudine;
    temp.latitudine = op1.latitudine + op2.latitudine;
    return temp;
  }
loc loc::operator=(loc op2)
  {
    longitudine = op2.longitudine;
    latitudine = op2.latitudine;
    return *this;
  }
void main(void)
  {
    loc ob1(10,20), ob2(5,30);
    ob1 = ob1+ob2;
    ob1.arata();
  }
```

953 RESTRICȚII LA SUPRAÎNCĂRCAREA OPERATORILOR CU FUNCȚII FRIEND

C/C++

Așa cum există restricții la supraîncărcarea operatorilor în cadrul claselor, C++ impune două restricții la supraîncărcarea operatorilor cu funcții *friend*. Prima restricție: trebuie să utilizați un parametru de referință către o clasă atunci când supraîncărcați operatorul de *incrementare* sau *decrementare* cu funcția *friend*. Secțiunea 954 explică în detaliu cum se utilizează funcția *friend* pentru a supraîncărca operatorul de *incrementare* sau *decrementare*. A doua restricție: nu puteți utiliza o funcție *friend* pentru a supraîncărca operatorii listați în tabelul 953.

Operatori restricționați	
=	()
[]	->

***Tabelul 953** Operatori pe care nu puteți să îi supraîncărcați cu o funcție **friend**.*

954 UTILIZAREA UNEI FUNCȚII FRIEND PENTRU SUPRAÎNCĂRCAREA OPERATORILOR ++ ȘI – –

C/C++

Dacă doriți să utilizați o funcție *friend* pentru a supraîncărca operatorii de incrementare și decrementare, trebuie să transmiteți operandul ca parametru de referință. Trebuie să transmiteți parametrul de referință pentru că, după cum ați învățat, funcțiile *friend* nu pot accesa pointerul *this*. În plus, trebuie să fiți siguri că transmiteți operandul ca parametru de referință – altfel, C++ va trata operandul ca un parametru *prin valoare* și nu va efectua operațiile dorite cu parametrul. În schimb, funcția de supraîncărcare a operatorului trebuie să modifice parametrul *prin referință*, înainte ca el să existe. Pentru a înțelege mai bine această prelucrare, analizați următorul program, *frn_inc.cpp*, care supraîncarcă operatorii de *incrementare* și *decrementare* pentru clasa *loc*:

```
#include <iostream.h>

class loc
  {
    int longitudine, latitudine;
   public:
    loc(void) {} // Utilizat pentru a construi temporare
    loc (int lg, int lt)
      {
        longitudine = lg;
        latitudine = lt;
      }
    void arata(void)
      {
        cout << longitudine << " ";
        cout << latitudine << endl;
      }
    loc operator=(loc op2);
    friend loc operator++(loc &op1); // Supraincarcare prin
```

```
                                             // functie friend
      friend loc operator--(loc &op1); // Supraincarcare prin
                                             // functie friend
   };
loc loc::operator=(loc op2)
   {
     longitudine = op2.longitudine;
     latitudine = op2.latitudine;
     return *this;
   }
loc operator++(loc &op)
   {
     op.longitudine++;
     op.latitudine++;
     return op;
   }
loc operator--(loc &op)
   {
     op.longitudine--;
     op.latitudine--;
     return op;
   }
void main(void)
   {
     loc ob1(10,20), ob2;
     ob1.arata();
     ++ob1;
     ob1.arata();      // Afiseaza 11 si 21
     ob2 = ++ob1;
     ob2.arata();      //Afiseaza 12 si 22
     --ob2;
     ob2.arata();      //Afiseaza din nou 11 si 21
   }
```

MOTIVE PENTRU SUPRAÎNCĂRCAREA OPERATORILOR CU FUNCȚII FRIEND

C/C++ 955

În multe cazuri, utilizarea fie a funcțiilor *friend*, fie a funcțiilor *membru* pentru supraîncărcarea unui operator nu provoacă diferențe funcționale în programul dumneavoastră. În timp ce supraîncărcarea cu o funcție *friend* nu este substanțial diferită de utilizarea unei funcții *membru* pentru supraîncărcare, ar trebui să utilizați funcții membre pentru a obține o încapsulare sporită. Însă, există unele situații, dintre care una în mod special, așa cum veți învăța în această secțiune, în care o funcție *friend* este extrem de folositoare.

După cum ați învățat, atunci când utilizați o funcție *membru* pentru a supraîncărca un operator *binar*, obiectul din partea stângă a operatorului generează apelarea funcției operator supraîncărcate. În plus, C++ transmite un pointer către obiectul din partea stângă în

cadrul pointerului *this*. De aceea, când creați o clasă numită *Pisici* și supraîncărcați operatorul plus, următoarea instrucțiune este validă, presupunând că ați creat o instanță obiect numită *happy*:

```
happy + 100
```

În exemplul precedent, *happy* generează apelarea funcției plus supraîncărcate, care va efectua adunarea și va returna valoarea în pointerul *this* la *happy*. Dacă însă scrieți expresia cum arătăm în exemplul următor, compilatorul va returna o eroare:

```
100 + happy
```

Deoarece funcția plus supraîncărcată așteaptă să primească un obiect clasă pe care îl poate referenția cu pointerul *this*, constanta pe care o primește în exemplul precedent provoacă eroare la compilare. Dacă, pe de altă parte, supraîncărcați operatorul plus cu o pereche de funcții *friend*, puteți realiza același lucru, fără a provoca o eroare. Următorul program, *doi_frd.cpp*, utilizează clasa *loc* pentru a arăta modul de utilizare a două funcții *friend*:

```
#include <iostream.h>

class loc {
      in longitudine, latitudine;
    public:
      loc(void) {} // Utilizat pentru a construi temporare
      loc (int lg, int lt)
        {
          longitudine = lg;
          latitudine = lt;
        }
      void arata(void)
        {
          cout << longitudine << " ";
          cout << latitudine << endl;
        }
      loc operator=(loc op2);
      friend loc operator+(loc op1, int op2); // Supraincarcare
                                              // prin functie
                                              // friend
      friend loc operator+(int op1, loc op2); // Supraincarcare
                                              // prin functie
                                              // friend
  };
loc loc::operator=(loc op2)
  {
    longitudine = op2.longitudine;
    latitudine = op2.latitudine;
    return *this;
  }
loc operator+(loc op1, int op2)
  {
```

```
    loc temp;
    temp.longitudine = op1.longitudine + op2;
    temp.latitudine = op1.latitudine + op2;
    return temp;
  }
loc operator+(int op1, loc op2)
  {
    loc temp;
    temp.longitudine = op1 + op2.longitudine;
    temp.latitudine = op1 + op2.latitudine;
    return temp;
  }
void main(void)
  {
    loc ob1(10,20), ob2( 5,30), ob3( 7,14);
    ob1.arata();
    ob2.arata();
    ob3.arata();
    ob1 = ob2 + 10;
    ob3 = 10 + ob3;
    ob1.arata();
    ob3.arata();
  }
```

SUPRAÎNCĂRCAREA OPERATORULUI NEW

C/C++ 956

După cum ați învățat, programele dumneavoastră pot să supraîncarce aproape orice funcție sau operator. Efectiv, programele dumneavoastră pot supraîncărca atât operatorul *new*, cât și operatorul *delete*. Aveți posibilitatea să alegeți supraîncărcarea oricăruia dintre acești operatori dacă doriți ca programele dumneavoastră să utilizeze o anumită metodă specială de alocare a memoriei. De exemplu, aveți posibilitatea să scrieți o rutină de alocare care utilizează hard-discul pentru memoria virtuală dacă programul ocupă toată memoria disponibilă din zona heap. Indiferent de motivul pentru care doriți supraîncărcarea funcției *new*, procedura este relativ simplă:

```
#include <stdlib.h>
void *operator new(size_t dimensiune)
  {
    // efectueaza alocarea
    return pointer_la_memorie;
  }
```

Tipul *size_t* trebuie să fie un tip capabil să păstreze partea cea mai mare din memorie pe care funcția supraîncărcată *new* o poate aloca. Fișierul antet *stdlib.h* definește tipul *size_t*. Parametrul *dimensiune* trebuie să conțină numărul de octeți pe care *new* îi cere pentru a păstra obiectul tocmai alocat. În sfârșit, funcția *new* trebuie să returneze un pointer la memoria alocată sau să returneze *NULL* dacă eșuează.

957 SUPRAÎNCĂRCAREA OPERATORULUI DELETE



În același mod în care puteți supraîncărca operatorul *new* pentru a controla necesitățile specifice de alocare de memorie, puteți supraîncărca și operatorul *delete* pentru a elibera memoria alocată de un operator supraîncărcat *new*. Operatorul *delete* trebuie să primească un pointer la memoria pe care operatorul *new* a alocat-o anterior pentru obiect. Puteți supraîncărca atât operatorul *new*, cât și *delete*, fie global, fie relativ la o clasă sau mai multe. Următorul program, *new_del.cpp*, utilizează funcțiile *new* și *delete* supraîncărcate pentru clasa *loc*:

```
#include <iostream.h>
#include <stdlib.h>

class loc {
    int longitudine, latitudine;
   public:
    loc(void) {} // Utilizat pentru a construi temporare
    loc (int lg, int lt)
      {
       longitudine = lg;
       latitudine = lt;
      }
    void arata(void)
      {
      cout << longitudine << " ";
      cout << latitudine << endl;
      }
    void *operator new(size_t dimensiune);
    void operator delete(void *p);
   };
void *loc::operator new(size_t dimensiune)
  {
   cout << "Functia new proprie." << endl;
   return malloc(dimensiune);
  }
void loc::operator delete(void *p)
  {
   cout << "Functia delete proprie." << endl;
   free(p);
  }
void main(void)
  {
  loc *p1, *p2;
  p1 = new loc(10,20);
  if (!p1)
   {
    cout << "Eroare la alocare\n";
    exit(1);
```

```
}
p2 = new loc(-10,-20);
if (!p2)
  {
    cout << "Eroare la alocare\n";
    exit(1);
  }
p1->arata();
p2->arata();
delete p1;
delete p2;
exit (0);
}
```

Când compilați și executați programul *new_del.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Functia new proprie.
Functia new proprie.
10 20
-10 -20
Functia delete proprie.
Functia delete proprie.
C:\>
```

SUPRAÎNCĂRCAREA OPERATORILOR NEW ȘI DELETE PENTRU MATRICE

C/C++ 958

În secțiunile precedente, ați învățat cum se supraîncarcă operatorii *new* și *delete* pentru a efectua alocări personalizate de memorie în cadrul programelor dumneavoastră. Dacă însă doriți să alocați matrice de obiecte, trebuie să supraîncărcați funcțiile *new* și *delete* din nou, utilizând un operator special prin care spune compilatorului că supraîncărcarea este pentru matrice. Prototipul acestei funcții *new* supraîncărcate pe care o puteți folosi pentru alocarea matricelor în cadrul programelor dumneavoastră este prezentată în continuare:

```
#include <stdlib.h>

void *operator new[](size_t dimensiune)
  {
    // efectueaza alocarea
    return pointer_la_memorie;
  }
```

Atunci când alocați matrice, C++ va apela automat funcția constructor a clasei pentru fiecare obiect al matricei. Când eliberați o matrice, C++ va apela automat funcția destructor a obiectului. Următorul program, *nd_tabl.cpp*, utilizează funcțiile supraîncărcate *new* și *delete* pentru a aloca și elibera spațiu pentru o matrice:

```
#include <iostream.h>
#include <stdlib.h>
```

```
class loc {
  int longitudine, latitudine;
  public:
    loc(void) {} // Utilizat pentru a construi temporare
    loc (int lg, int lt)
      {
        longitudine = lg;
        latitudine = lt;
      }
    void arata(void)
      {
        cout << longitudine << " ";
        cout << latitudine << endl;
      }
    void *operator new(size_t dimensiune);
    void operator delete(void *p);
    void *operator new[](size_t dimensiune);
    void operator delete[](void *p);
  };
void *loc::operator new(size_t dimensiune)
  {
    cout << "Functia new proprie." << endl;
    return malloc(dimensiune);
  }
void loc::operator delete(void *p)
  {
    cout << "Functia delete proprie." << endl;
    free(p);
  }
void *loc::operator new[](size_t dimensiune)
  {
    cout << "Functia new proprie pentru alocarea matricei." << endl;
    return malloc(dimensiune);
  }
void loc::operator delete[](void *p)
  {
    cout << "Elibereaza matricea cu functia delete proprie."
      << endl;
    free(p);
  }
void main(void)
  {
    loc *p1, *p2;
    int i;
    p1 = new loc(10,20);
    if (!p1)
      {
```

```
      cout << "Eroare la alocare\n";
      exit(1);
    }
  p2 = new loc[10];
  if (!p2)
    {
      cout << "Eroare la alocare\n";
      exit(1);
    }
  p1->arata();
  for(i=0; i<10; i++)
    p2[i].arata();
  delete p1;
  delete [] p2;
}
```

Programul *nd_tabl.cpp* supraîncarcă operatorii *new* și *delete* atât pentru matrice cât și pentru obiecte individuale. Atunci când programul creează o instanță matrice a unui obiect, el invocă operatorul personalizat *new* pentru matrice și operatorul personalizat *new* pentru fiecare element al matricei. Programul efectuează prelucrări similare atunci când șterge o instanță individuală sau o matrice. Când compilați și executați programul *nd_tabl.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Functia new proprie.
Functia new proprie pentru alocarea matricei.
10 20
4258096 4258096
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
Functia delete proprie.
Elibereaza matricea cu functia delete proprie.
C:\>
```

SUPRAÎNCĂRCAREA OPERATORULUI DE MATRICE [ ]

C/C++ 959

Pe măsură ce programele dumneavoastră devin mai complexe, uneori va trebui să supraîncărcați operatorul de *matrice* [ ]. C++ consideră operatorul de matrice ca pe un operator *binar* pentru scopurile supraîncărcării. Prin urmare, forma generală a supraîncărcării unei funcții membre operator este:

```
tip nume-clasa::operator[ ](int i)
   {
     // . . .
   }
```

Din punct de vedere tehnic, parametrul *i* din exemplul precedent nu trebuie neapărat să fie de tipul *int*, dar deoarece dumneavoastră veți defini de obicei matricele cu un parametru întreg, trebuie să evitați utilizarea unui parametru de tipul *float* sau de alt tip. Atunci când apelați funcția operator supraîncărcată, C++ va atribui pointerul *this* la obiect și va folosi parametrul pentru a controla dimensiunea. Pentru a înțelege mai bine prelucrările pe care le efectuează funcția supraîncărcată [ ] pentru matrice, analizați următorul program, *tabl_supr.cpp*:

```
#include <iostream.h>

class tipoarecare {
     int a[3];
   public:
     tipoarecare(int i, int j, int k)
       {
         a[0] = i;
         a[1] = j;
         a[2] = k;
       }
     int operator[ ](int i) {return a[i];}
   };
void main(void)
   {
     tipoarecare ob(1, 2, 3);
     cout << ob[1];
   }
```

Supraîncărcarea operatorului [ ] pentru matrice vă oferă posibilitatea de a controla mai bine crearea matricelor cu clase. Pe lângă faptul că vă permite să atribuiți valori distincte pentru fiecare membru, puteți utiliza funcțiile supraîncărcate pentru a crea un program care să efectueze o indexare sigură a unei matrice. Indexarea sigură a unei matrice contribuie la prevenirea supradepășirii sau subdepășirii limitelor unei matrice în decursul execuției unui program. Următorul program, *sig_tabl.cpp*, extinde programul *tabl_supr.cpp* pentru a realiza și indexarea sigură a matricei:

```
#include <iostream.h>
#include <stdlib.h>

class tipoarecare {
     int a[3];
   public:
     tipoarecare(int i, int j, int k)
       {
         a[0] = i;
         a[1] = j;
```

```
        a[2] = k;
    }
    int &operator[ ](int i);
};
int &tipoarecare::operator[ ](int i)
{
    if (i<0 || i>2)
    {
        cout << "Eroare la margini.\n";
        exit(1);
    }
    return a[i];
}
void main(void)
{
    tipoarecare ob(1, 2, 3);
    cout << ob[1];
    cout << endl;
    ob[1] = 25;
    cout << endl;
    cout << ob[1];
    ob[3] = 44;
}
```

Atunci când încercați să accesați un obiect de dincolo de marginile matricei, se va produce o eroare. În cazul programului *sig_tabl.cpp*, încercarea de a accesa elementul de la indicele 3 se face dincolo de margini și prin urmare programul va returna o eroare. Atunci când executați programul *sig_tabl.cpp*, el va genera următorul rezultat:

```
2
25
Eroare la margini
C:\>
```

SUPRAÎNCĂRCAREA OPERATORULUI APEL DE FUNCȚIE ()

C/C++ 960

După cum ați învățat, C++ vă permite să supraîncărcați mare parte din operatorii săi în decursul unui program. Atunci când supraîncărcați operatorul *apel de funcție* (), nu înseamnă că veți crea o nouă metodă de a apela o funcție. În schimb, creați o funcție *operator* către care programele dumneavoastră pot transmite un număr arbitrar de parametri. În general, atunci când supraîncărcați operatorul apel de funcție (), definiți parametrii pe care vreți ca programul să îi transmită funcției supraîncărcate. Pentru a înțelege mai bine modul în care C++ supraîncarcă operatorul apel de funcție (), analizați următorul program, *funsupra.cpp*, care utilizează operatorul *apel de funcție* supraîncărcat cu clasa *loc*:

```
#include <iostream.h>
class loc {
```

```
    int longitudine, latitudine;
   public:
     loc(void) {} // Utilizat pentru a construi temporare
     loc (int lg, int lt)
       {
         longitudine = lg;
         latitudine = lt;
       }
     void arata(void)
       {
         cout << longitudine << " ";
         cout << latitudine << endl;
       }
     loc operator+(loc op2);
     loc operator()(int i, int j);
   };
loc loc::operator()(int i, int j)
   {
     longitudine = i;
     latitudine = j;
     return *this;
   }
loc loc::operator+(loc op2)
   {
     loc temp;
     temp.longitudine = op2.longitudine + longitudine;
     temp.latitudine = op2.latitudine + latitudine;
     return temp;
   }
void main(void)
   {
     loc ob1(10,20), ob2( 1,1);
     ob1.arata();
     ob1(7,8);
     ob1.arata();
     ob1 = ob2 + ob1(10,10);
     ob1.arata();
   }
```

În programul *funsupra.cpp*, operatorul *apel de funcție* () supraîncărcat pentru clasa *loc* vă permite atribuirea de noi valori unui obiect urmat de operatorul *apel de funcție* (). În program, penultima instrucțiune, *ob1 = ob2 + ob1(10,10)*, beneficiază de avantajele operatorului *apel de funcție ()* supraîncărcat pentru a atribui în mod dinamic o valoare lui *ob1*. În acest exemplu particular, programul nu păstrează valorile de curând atribuite. Totuși, dacă ați utiliza operatorul *apel de funcție ()* cu un obiect diferit, să spunem *ob3*, programul ar păstra valoarea nou atribuită în acel obiect.

Atunci când compilați și executați programul *funsupra.cpp*, ecranul dumneavoastră va afișa următoarele:

```
10 20
7 8
11 11
C:\>
```

SUPRAÎNCĂRCAREA OPERATORULUI POINTER ->

C/C++ 961

După cum ați învățat, C++ vă permite supraîncărcați mulți dintre operatorii săi. Pe măsură ce programele dumneavoastră devin mai complexe, uneori va trebui să supraîncărcați operatorul *pointer*. În aceste situații, trebuie mai întâi să înțelegeți că C++ tratează operatorul *pointer* ca pe un operator unar (deci, un operator cu numai un singur operand) când îl supraîncărcați. Atunci când supraîncărcați funcția *pointer*, trebuie să returnați un pointer către un obiect al clasei apelante. Când supraîncărcați operatorul *pointer* ->, valoarea sa returnată este aceeași cu cea pe care programul ar trebui să o primească dacă ar invoca operatorul punct cu obiectul. Cu alte cuvinte, următoarele instrucțiuni sunt echivalente:

```
ob->i = 10;
ob.i = 10;
```

Pentru a înțelege mai bine modul în care C++ efectuează supraîncărcarea operatorilor *pointer* ->, să analizăm următorul program, *supraptr.cpp*, prezentat în continuare:

```
#include <iostream.h>

class exemplu {
  public:
   int i;
   exemplu *operator->(void) {return this;}
  };
void main(void)
  {
   exemplu ob;
   ob->i = 10;        // Acelasi cu ob.i
   cout << ob.i << " " << ob->i;
  }
```

Observație: *Pare că nu există cu adevărat un scop util pentru a supraîncărca operatorul pointer->. Totuși, dacă veți descoperi că trebuie să faceți acest lucru, veți utiliza forma prezentată în această secțiune.*

SUPRAÎNCĂRCAREA OPERATORULUI VIRGULĂ ,

C/C++ 962

După cum ați învățat, programele dumneavoastră pot supraîncărca mulți dintre operatorii C++. Cum programele și clasele dumneavoastră devin mai complexe, puteți descoperi că programele trebuie să supraîncarce operatorul *virgulă*. În C++, operatorul *virgulă* evaluează fiecare operand din liste separate prin virgulă și returnează numai ultimul operand din dreapta listei.

Cu alte cuvinte, când codul dumneavoastră are o listă *E1, E2*, programul va evalua operandul stâng *E1* ca o expresie *void* și va returna evaluarea lui *E2* ca fiind rezultatul și tipul expresiei virgulă. În mod recursiv, operatorul virgulă are ca rezultat evaluarea expresiei *E1, E2, ..., En* de la stânga la dreapta. Operatorul virgulă evaluează fiecare *Ei* pe rând și returnează valoarea și tipul lui *En* ca rezultat al întregii expresii. Pentru a evita ambiguitatea dintre operatorul virgulă și virgula delimitatoare în argumentele funcției și listele de inițializare, utilizați parantezele, ca mai jos:

```
func(i, (j = 1, j + 4), k);
```

Fragmentul de cod precedent apelează *func* cu trei argumente *(i, 5, k)*, nu patru. În timp ce programele dumneavoastră pot supraîncărca operatorul *virgulă* în orice fel doriți, ar trebui să încercați să mențineți consistența cu operația implicită din C++ pentru virgulă. Pentru a înțelege mai bine, analizați următorul program, *virgula.cpp*, care supraîncarcă operatorul *virgulă*, dar menține operația sa normală:

```
#include <iostream.h>

class loc {
   int longitudine, latitudine;
   public:
     loc(void) {} // Utilizat pentru a construi temporare
     loc (int lg, int lt)
       {
         longitudine = lg;
         latitudine = lt;
       }
     void arata(void)
       {
         cout << longitudine << " ";
         cout << latitudine << endl;
       }
     loc operator+(loc op2);
     loc operator,(loc op2);
   };
loc loc::operator,(loc op2)
   {
     loc temp;
     temp.longitudine = op2.longitudine;
     temp.latitudine = op2.latitudine;
     cout << op2.longitudine << " " << op2.latitudine << endl;
     return temp;
   }
loc loc::operator+(loc op2)
   {
     loc temp;
     temp.longitudine = op2.longitudine + longitudine;
     temp.latitudine = op2.latitudine + latitudine;
     return temp;
```

```
}
void main(void)
  {
    loc ob1(10,20), ob2( 5,30), ob3(1,1);
    ob1.arata();
    ob2.arata();
    ob3.arata();
    cout << endl;
    ob1 = (ob1, ob2 + ob2, ob2 + ob3);
    ob1.arata(); // Va afisa 6,31, valorile lui ob2 + ob3
  }
```

Atribuirea listei asupra căreia a operat virgula în penultima linie din programul *virgula.cpp*, se efectuează ca mai jos:

```
ob1 = (ob1, ob2 + ob2, ob2 + ob3);
ob1 = ob1;
ob1 = ob2 + ob2;
ob1 = ob2 + ob3;
ob1 = (6, 31);
```

ABSTRACTIZAREA

C/C++ 963

Abstractizarea este procesul de abordare a unui obiect prin prisma metodelor sale (operații), în timp ce se ignoră temporar detalii elementare ale implementării obiectului. Programatorii utilizează abstractizarea pentru a simplifica proiectul și implementarea programelor complexe. De exemplu, dacă doriți să scrieți un program procesor de texte, sarcina ar părea la început foarte dificilă. Însă, utilizând abstractizarea, veți începe să înțelegeți că un procesor de texte constă de fapt din obiecte, cum ar fi obiecte document pe care le veți crea, salva, verifica ortografic și tipări. Privind programul în termeni de abstractizare, veți putea să înțelegeți mai bine solicitările programului. În C++, cel mai important instrument pentru susținerea abstractizării este clasa.

ALOCAREA UNUI POINTER LA O CLASĂ

C/C++ 964

Pe măsură ce lucrați cu variabile de clasă, puteți să alocați matrice dinamice sau liste dinamice de tipul clasei. După cum ați învățat, puteți utiliza matrice dinamice și liste dinamice în cadrul programelor dumneavoastră atunci când, la momentul compilării, nu cunoașteți numărul de elemente pe care îl va solicita matricea sau lista. Declararea unei matrice dinamice de obiecte clasă este de principiu identică cu declararea unei matrice dinamice de orice tip de bază din C sau C++. Următorul program, *din_clas.cpp*, de exemplu, creează o matrice de pointeri la variabilele unei clase de tip *Carte*:

```
#include <iostream.h>
#include <<iomanip.h>
#include <<string.h>

class Carte
```

```
   {
     public:
       void arata_titlu(void) { cout << titlu << '\n'; };
       void arata_carte(void)
         {
           arata_titlu();
           arata_editura();
         };
       Carte(char *titlu, char *autor, char *editura, float pret);
     private:
       char titlu[256];
       char autor[64];
       float pret;
       char editura[256];
       void arata_editura(void) { cout << editura << '\n'; };
   };
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
   {
     strcpy(Carte::titlu, titlu);
     strcpy(Carte::autor, autor);
     strcpy(Carte::editura, editura);
     Carte::pret = pret;
     cout << "Functia constructor." << endl;
   }
void main(void)
   {
     Carte *Biblioteca[4];
     int i;
     Biblioteca[0] = new Carte("Jamsa's C/C++ Programmer's Bible",
                        "Jamsa & Klander", "Jamsa Press", 49.95);
     Biblioteca[1] = new Carte("Hacker Proof", "Klander",
                               "Jamsa Press", 54.95);
     Biblioteca[2] = new Carte("ActiveX Programmer's Library",
            "Lalani & Chandak", "Jamsa Press", 49.95);
     Biblioteca[3] = new Carte("Rescued by C++, "Jamsa",
                               "Jamsa Press", 29.95);
     for (i = 0; i < 4; i++)
       Biblioteca[i]->arata_carte();
   }
```

Când compilați și executați programul *din_clas.cpp*, ecranul dumneavoastră va afișa următoarea ieșire:

```
Functia constructor.
Functia constructor.
Functia constructor.
Functia constructor.
Jamsa's C/C++ Programmer's Bible
Jamsa Press
```

```
Hacker Proof
Jamsa Press
ActiveX Programmer's Library
Jamsa Press
Rescued by C++
Jamsa Press
C:\>
```

După cum vedeți, de fiecare dată când creați o instanță utilizând operatorul *new*, C++ invocă funcția constructor a clasei.

ELIMINAREA UNUI POINTER LA O CLASĂ

C/C++ 965

În secțiunea 964 ați creat o matrice de pointeri la obiecte de tip *Carte*. De fiecare dată când programul creează o instanță, C++ va invoca automat funcția constructor a clasei *Carte*. În mod similar, dacă acea clasă are un destructor, C++ va invoca automat funcția destructor de fiecare dată când programul distruge o instanță. Următorul program, *dindestr.cpp*, adaugă o funcție destructor clasei *Carte*. Programul utilizează, de asemenea, operatorul *delete* pentru a elimina pointerul la fiecare instanță, cum se prezintă în continuare:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Carte
  {
    public:
      void arata_titlu(void) { cout << titlu << '\n'; };
      void arata_carte(void)
        {
          arata_titlu();
          arata_editura();
        };
      Carte(char *titlu, char *autor, char *editura, float pret);
      ~Carte(void) { cout << "Distruge intrarea pentru "
         << titlu <<
          endl; };
    private:
      char titlu[256];
      char autor[64];
      float pret;
      char editura[256];
      void arata_editura(void) { cout << editura << '\n'; };
  };
Carte::Carte(char *titlu, char *autor, char *editura, float pret)
  {
    strcpy(Carte::titlu, titlu);
    strcpy(Carte::autor, autor);
    strcpy(Carte::editura, editura);
    Carte::pret = pret;
```

```
   }
void main(void)
   {
     Carte *Biblioteca[4];
     int i = 0;
     Biblioteca[0] = new Carte("Jamsa's C/C++ Programmer's Bible",
       "Jamsa & Klander", "Jamsa Press", 49.95);
     Biblioteca[1] = new Carte("Hacker Proof", "Klander",
                                  "Jamsa Press", 54.95);
     Biblioteca[2] = new Carte("ActiveX Programmer's Library",
       "Lalani & Chandak", "Jamsa Press", 49.95);
     Biblioteca[3] = new Carte("Rescued by C++, "Jamsa",
                                  "Jamsa Press", 24.95);
     for (i = 0; i < 4; i++)
       Biblioteca[i]->arata_carte();
     for (i = 0; i < 4; i++)
       delete Biblioteca[i];
   }
```

Atunci când compilaţi şi executaţi programul *dindestr.cpp*, ecranul dumneavoastră va afişa următoarea ieşire:

```
Jamsa's C/C++ Programmer's Bible
Jamsa Press
Hacker Proof
Jamsa Press
ActiveX Programmer's Library
Jamsa Press
Rescued by C++
Jamsa Press
Distruge intrarea pentru Jamsa's C/C++ Programmer's Bible
Distruge intrarea pentru Hacker Proof
Distruge intrarea pentru ActiveX Programmer's Library
Distruge intrarea pentru Rescued by C++
C:\>
```

966 ELIMINAREA SPAŢIULUI ALB CARE PRECEDE O INTRARE

C/C++

După cum aţi învăţat, fluxul de intrare/ieşire *cin* utilizează spaţiul alb ca delimitator pentru datele de intrare. Atunci când utilizaţi *cin*, e posibil ca *cin* să ignore spaţiul alb cu care precede textul. În astfel de cazuri, programele dumneavoastră pot utiliza manipulatorul *ws*, ca mai jos:

```
cin >> ws >> buffer;
```

Următorul program, *ws.cpp*, utilizează manipulatorul *ws* pentru a elimina spaţiul alb care precede intrarea:

```
#include <iostream.h>

void main(void)
```

```
{
  char buffer[256];
  cout << "Introduce un cuvant precedat de spatiul alb" << endl;
  cin >> ws >> buffer;
  cout << "==" << buffer << "==";
}
```

Pentru a testa programul *ws.cpp*, eliminați manipulatorul *ws* și modificați intrarea precedată de spațiul alb. De exemplu, atunci când compilați și executați programul *ws.cpp* și introduceți cuvântul „Jamsa", ieșirea va fi aceeași ca în cazul în care ați fi introdus cuvântul „Jamsa":

```
Introduce un cuvant precedat de spatii albe
 Jamsa
==Jamsa==
C:\>ws
Introduce un cuvant precedat de spatii albe
Jamsa
==Jamsa==
C:\>
```

BIBLIOTECILE DE CLASE

C/C++ 967

După cum ați învățat, bibliotecile de obiecte fac ca reutilizarea funcțiilor să fie foarte ușoară pentru programele dumneavoastră. O *bibliotecă de clase* este similară cu o bibliotecă de obiecte prin aceea că ea conține codul la care programul dumneavoastră poate să se lege. Spre deosebire de biblioteca de cod obiect, care conține o colecție de funcții apelabile, o bibliotecă de clase conține metode de clasă. Pentru a utiliza metodele, programele dumneavoastră trebuie să utilizeze structurile de clasă corespunzătoare. Cu alte cuvinte, programele dumneavoastră nu pot să apeleze pur și simplu funcții ale bibliotecii de clase, fără să utilizeze o clasă. Pe parcursul acestei cărți, programele dumneavoastră au utilizat frecvent biblioteca de clase C++ *iostream* pentru a efectua operații de I/O, utilizând *cin* și *cout*. La fel cum puteți crea biblioteci de cod obiect care conțin funcții create de dumneavoastră, puteți, de asemenea, să creați biblioteci de clase. Prin crearea propriilor dumneavoastră biblioteci de clase, veți putea utiliza ușor obiectele existente în programele dumneavoastră viitoare.

PLASAȚI DEFINIȚIILE CLASELOR DUMNEAVOASTRĂ ÎN FIȘIERE ANTET

C/C++ 968

Atunci când creați o clasă pe care o poate utiliza și alt programator, ar trebui să plasați declarația clasei într-un fișier antet bazat pe numele clasei. De exemplu, fișierul antet *iostream.h* conține declarația de clasă pentru clasa *iostream*. Nu plasați metodele clasei în fișierul antet. În schimb, compilați metodele clasei și plasați-le într-o bibliotecă de clase, așa cum ați învățat în secțiunea de mai sus. Prin plasarea declarației de clasă în fișiere antet, faceți mai simplă utilizarea clasei de către program. Programul nu trebuie să cunoască structura completă a clasei, cât trebuie să includă fișierul antet al clasei și apoi să folosească numai acei membri de care are nevoie.

969 UTILIZAREA CUVÂNTULUI CHEIE INLINE CU FUNCȚIILE MEMBRE ALE CLASEI



După cum știți, cuvântul cheie *inline* spune compilatorului să plaseze un cod de funcție *inline* la fiecare referință. Utilizarea cuvântului cheie *inline* vă permite să faceți un echilibru între creșterea dimensiunii programului și îmbunătățirea performanței lui. Atunci când definiți o funcție membru al unei clase, C++ vă permite să plasați funcțiile în cadrul clasei înseși sau în afara acesteia. Când plasați o definiție a funcției în interiorul clasei, C++ generează cod *inline* de fiecare dată când întâlnește o invocare a respectivei metode. Dacă aveți o metodă care este definită în afara clasei pentru care doriți să se genereze cod *inline*, precedați pur și simplu numele funcției cu cuvântul cheie *inline*. De exemplu, următoarea definiție de funcție cere compilatorului să genereze cod *inline* pentru fiecare invocare a metodei *arata_carte*:

```
inline void Carte::arata_carte(void) { arata_titlu();
  arata_editura(); };
```

970 INIȚIALIZAREA UNEI MATRICE DE CLASE



După cum ați învățat, C++ vă permite declararea unei matrice de clase. Atunci când declarați o matrice de clase, programul va invoca automat funcția constructor pentru fiecare element. Când declarați o matrice de structuri, C++ vă permite inițializarea membrului matrice, ca mai jos:

```
#struct Angajat
  {
    char nume[64];
    long id;
  } muncitori [2] = {{"Kris", 1}{"Happy, 2"}};
```

Atunci când declarați o matrice de clase, C++ nu vă permite să dispuneți de valori inițiale. De aceea, ați putea considera atribuirea de valori membrilor o încercare dificilă. Următorul program, *atrbtab.cpp*, utilizează funcția constructor pentru a atribui fiecare element al matricei:

```
#include <iostream.h>
#include <string.h>

class Angajat
  {
    public:
      Angajat(void);
      void arata_angajat(void) { cout << nume << endl; };
    private:
      char nume[256];
      long id;
  };
Angajat::Angajat(void)
  {
```

```
    static int index = 0;
    switch (index++) {
     case 0: strcpy(Angajat::nume, "Kris");
             Angajat::id = 1;
             break;
     case 1: strcpy(Angajat::nume, "Happy");
             Angajat::id = 2;
             break;
    };
  }
void main(void)
  {
    Angajat muncitori[2];
    muncitori[0].arata_angajat();
    muncitori[1].arata_angajat();
  }
```

Funcția constructor utilizează variabila statică *index* pentru a determina care element se inițializează. După cum vă puteți da seama, constructorul ar putea fi complet dezorientat, în raport de numărul elementelor și membrilor clasei.

DISTRUGEREA UNEI MATRICE DE CLASE

C/C++ 971

După cum ați învățat, programele dumneavoastră pot supraîncărca operatorul *delete* pentru a elibera zona heap de memoria unei matrice de clase. Ați învățat, de asemenea, că programele dumneavoastră vor utiliza cel mai frecvent funcțiile destructor pentru a elibera memoria sau a salva pe disc informații despre o clasă. Atunci când utilizați un operator supraîncărcat *delete* pentru a elibera o matrice, operatorul *delete* va apela funcția destructor pentru fiecare element din cadrul matricei. Când scrieți cod în cadrul funcției destructor, aveți grijă să optimizați pe cât posibil codul pentru a evita încetinirea programului în secvența distrugerii matricei. Pentru a înțelege mai bine codul special al destructorului pe care trebuie să îl utilizați cu matricea de clase, analizați programul *dis_tab.cpp*, care adaugă o funcție destructor la exemplul de program scris în secțiunea 958, ca mai jos:

```
#include <iostream.h>
#include <stdlib.h>
class loc
  {
     int longitudine, latitudine;
    public:
     loc(void) {} // Utilizat pentru a construi temporare
     ~loc(void);
     loc (int lg, int lt)
       {
         longitudine = lg;
         latitudine = lt;
       }
     void arata(void)
```

```
        {
          cout << longitudine << " ";
          cout << latitudine << endl;
        }
      void *operator new(size_t dimensiune);
      void operator delete(void *p);
      void *operator new[ ](size_t dimensiune);
      void operator delete[ ](void *p);
  };
loc::~loc(void)
  {
    cout << "Functia destructor" << endl;
  }
void *loc::operator new(size_t dimensiune)
  {
    cout << "Functia operator new personalizata." << endl;
    return malloc(dimensiune);
  }
void loc::operator delete(void *p)
  {
    cout << "Functia operator delete personalizata." << endl;
    free(p);
  }
void *loc::operator new[ ](size_t dimensiune)
  {
    cout << "Functia de alocare new personalizata a matricei."
      << endl;
    return malloc(dimensiune);
  }
void loc::operator delete[ ](void *p)
  {
    cout << "Eliberarea matricei cu functia delete personalizata."
      << endl;
    free(p);
  }
void main(void)
  {
    loc *p1, *p2;
    int i;
    p1 = new loc(10,20);
    if (!p1)
      {
        cout << "Eroare de alocare\n";
        exit(1);
      }
    p2 = new loc[10];
    if (!p2)
```

```
      {
        cout << "Eroare de alocare\n";
        exit(1);
      }
    p1->arata();
    for(i=0; i<10; i++)
      p2[i].arata();
    delete p1;
    delete [ ] p2;
}
```

Când executați programul *dis_tab.cpp*, programul va invoca destructorul pentru fiecare element al matricei de clase și apoi va apela destructorul personalizat al matricei, iar ecranul dumneavoastră va afișa următoarele:

```
Functia destructor
Functia operator delete personalizata.
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Functia destructor
Eliberarea matricei cu functia delete personalizata.
```

Atunci când programul apelează funcția *delete* personalizată, ea va activa imediat funcția destructor pentru fiecare obiect din matrice, înainte de a șterge pointerul la matrice. De aceea, puteți utiliza funcțiile destructor cu matrice pentru a păstra valori, pentru a genera noi informații sau a elimina valori din obiect.

CREAREA UNOR MATRICE DE CLASE INIȚIALIZATE

C/C++ 972

După cum ați învățat, C++ vă permite să creați matrice de obiecte. Atunci când creați o matrice de obiecte, puteți inițializa obiectele în două modalități: fie după crearea obiectelor (prin utilizarea unei bucle *for* pentru a trece prin matrice și a inițializa fiecare element din cadrul său), fie cu o funcție constructor a obiectului. De exemplu, următorul program, *simp_ini.cpp*, utilizează o buclă *for* pentru a inițializa o matrice de variabile obiecte de tip *exemplu*:

```
#include <iostream.h>
class exemplu
  {
    int valoare;
  public:
    void set_valoare(int j) {valoare = j;}
    int reda_valoare() {return valoare;}
```

```
   };
void main(void)
   {
     exemplu ob[3];
     int bucla;
     for(bucla=0; bucla<3; bucla++)
       ob[bucla].set_valoare(bucla+1);
     for(bucla=0; bucla<3; bucla++)
       cout << ob[bucla].reda_valoare() << endl;
   }
```

Atunci când compilaţi şi executaţi programul *simp_ini.cpp*, ecranul dumneavoastră va afişa următoarele:

```
1
2
3
C:\>
```

După cum puteţi vedea, programul ciclează prin matrice, iniţializând fiecare element. Apoi, programul ciclează prin matrice din nou, afişând fiecare element. În timp ce buclele *for* sunt utile când matricea este de mici dimensiuni, este mai potrivit să iniţializaţi valorile matricei dumneavoastră în cadrul declaraţiei ei, prin aceasta simplificându-vă programul. Următorul program, *unu_ini.cpp*, utilizează aceeaşi matrice din programul *simp_ini.cpp* prezentat anterior, dar iniţializează matricea în cadrul constructorului:

```
#include <iostream.h>

class exemplu
   {
     int valoare;
public:
     exemplu(int j) {valoare = j;} // constructor
     int reda_valoare(void) {return valoare;}
   };
void main(void)
   {
     int bucla;
     exemplu ob[3] = {1, 2, 3}; // initializator
     for(bucla=0; bucla<3; bucla++)
       cout << ob[bucla].reda_valoare() << endl;
   }
```

Atunci când compilaţi şi executaţi programul *unu_ini.cpp*, acesta va afişa aceeaşi ieşire cu programul *simp_ini.cpp*:

```
1
2
3
C:\>
```

După cum puteţi vedea, programul *unu_ini.cpp* este mai clar şi mai scurt decât programul *simp_ini.cpp*. De regulă, ar trebui să iniţializaţi matricele în cadrul constructorilor dacă ştiţi

valorile inițiale ale matricei. În secțiunea 974 veți învăța cum se scrie o clasă care acceptă matrice atât inițializate, cât și neinițializate.

INIȚIALIZAREA UNEI MATRICE CU UN CONSTRUCTOR CU MAI MULTE ARGUMENTE

C/C++973

În secțiunea 972 ați învățat cum se inițializează o matrice de obiecte în cadrul unei funcții constructor a obiectului. Exemplul simplu prezentat în secțiunea 972 inițializează numai cu o singură valoare. Însă, cele mai multe clase pe care le creați vor conține mai multe date membre pe care programul să le inițializeze automat. Următorul program, *doi_ini.cpp*, inițializează o matrice care conține două valori în cadrul clasei:

```
#include <iostream.h>
class exemplu
  {
     int valoare1;
     int valoare2;
   public:
     exemplu(int j, int k) // constructor
       {
         valoare1 = j;
         valoare2 = k;
       }
     int reda_valoare2() {return valoare2;}
     int reda_valoare1() {return valoare1;}
  };
void main(void)
  {
   exemplu ob[3] = {exemplu(1,2),
                   exemplu(3,4),
                   exemplu(5,6) }; // initializator
   int bucla;
   for(bucla=0; bucla<3; bucla++)
     {
       cout << "Valoare1, Valoare 2: ";
       cout << ob[bucla].reda_valoare1();
       cout << ", ";
       cout << ob[bucla].reda_valoare2() << endl;
     }
  }
```

Când compilați și executați programul *doi_ini.cpp*, ecranul dumneavoastră va afișa următoarea ieșire:

```
Valoare 1, Valoare 2: 1, 2
Valoare 1, Valoare 2: 3, 4
Valoare 1, Valoare 2: 5, 6
```

974 CREAREA UNEI MATRICE INIȚIALIZATE SAU CREAREA UNEI MATRICE NEINIȚIALIZATE

C/C++

Pe măsură ce programele dumneavoastră devin mai complexe, veți considera adesea că programele dumneavoastră trebuie să inițializeze anumite matrice ale unui obiect, dar nu în mod necesar și alte matrice. De exemplu, puteți inițializa două matrice pe care programul dumneavoastră le va utiliza și va lăsa neinițializată o a treia pe care programul o va utiliza numai pentru stocare temporară. Puteți ușor să supraîncărcați funcția constructor de clasă astfel încât clasele dumneavoastră să accepte atât declarațiile inițializate, cât și pe cele neinițializate. Următorul program, *inineini.cpp*, modifică proiectul clasei *exemplu* pentru a accepta ambele tipuri de declarații:

```
#include <iostream.h>
class exemplu
   {
      int valoare;
    public:
      exemplu() {valoare = 0;}; // constructor neinitializat
      exemplu(int j) {valoare = j;} // constructor initializat
      int reda_valoare() {return valoare;}
   };
void main(void)
   {
     int bucla;
     exemplu ob1[3] = {1, 2, 3};
     exemplu ob2[32];
     cout << "Introducerea primei bucle: " << endl;
     for(bucla=0; bucla<3; bucla++)
       cout << ob1[bucla].reda_valoare() << endl;
     cout << "Introducerea celei de a doua bucle: " << endl;
     for(bucla=0; bucla<32; bucla++)
       cout << ob2[bucla].reda_valoare() << ", ";
     cout << endl;
   }
```

După cum vedeți, programul *inineini.cpp* definește ambii constructori: cel implicit, neinițializat și un constructor inițializat. Definițiile din *main* creează o matrice de trei obiecte inițializate și o matrice de 32 de obiecte neinițializate. Atunci când programul ciclează prin matrice, prima matrice va afișa valorile la care programul a inițializat la început elementele. O a doua matrice, însă, va afișa 0 pentru fiecare element, deoarece constructorul implicit inițializează fiecare element la 0. Când compilați și executați programul *inineini.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Introducerea primei bucle:
1
2
3
Introducerea celei de a doua bucle:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

LUCRUL CU MATRICELE DE CLASE

C/C++ 975

Atunci când lucrați cu matrice de clase, veți trata matricea de clase asemănător cu matricele de structuri atunci când ați lucrat cu ele în C. După cum știți, matricele și clasele vă permit să grupați informațiile corelate. Așa cum ați învățat, C++ vă permite să creați matrice de obiecte sau să utilizați matrice ca membri ai clasei. În general, C++ nu fixează o limită a adâncimii la care programele dumneavoastră pot ajunge, în ceea ce privește structurile de date imbricate. De exemplu, următoarea declarație creează o matrice de 100 de obiecte *angajat*. În cadrul fiecărui element există o matrice de structuri *Date* care corespunde datei de angajare, a primei și a ultimei examinări:

```
class angajat
  {
    public:
      char nume[64];
      int varsta;
      char nrss[11]; // Numarul de asigurare sociala
      int grad_salariz;
      float salariu;
      unsigned cod_angajat;
      struct Date
        {
          int luna;
          int zi;
          int an;
        } date_angaj[3];
      int date_examin(date curente)
  };
// instructiunile programului
void main(void);
  {
    angajat membrii_conducere[100];
    // codul programului
  }
```

Pentru a accesa membrii și elementele matricei, pur și simplu veți lucra de la dreapta la stânga, începând din afară și mergând spre interior. De exemplu, următoarea instrucțiune atribuie data angajării unui salariat:

```
membrii_conducere[10].data_angaj[0].luna = 10;
membrii_conducere[10].data_angaj[0].zi = 31;
membrii_conducere[10].data_angaj[0].an = 97;
```

Cu toate că imbricarea obiectelor și a matricelor arătată în această secțiune este adeseori mai convenabilă, rețineți că, pe măsură ce programele dumneavoastră vor utiliza structuri de date mai complex imbricate, ele vor deveni mai greu de înțeles pentru alți programatori.

Observație: *Pe lângă faptul că definiția clasei* ***angajat*** *din această secțiune confirmă neclaritatea claselor imbricate, ea nici nu încapsulează bine clasa. Așa cum ați învățat, programul dumneavoastră trebuie să utilizeze în general funcții de interfață pentru a*

manipula și a returna informații în cadrul clasei. De fapt, încapsularea efectivă a datelor face clasele imbricate mai dificil de folosit.

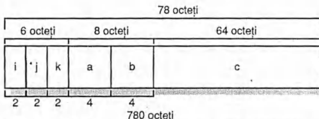
976 CUM MANEVREAZĂ MEMORIA MATRICELE DE CLASE

C/C++

După cum ați învățat, în ciuda cantităților suficiente de memorie disponibile pentru programele dumneavoastră atunci când lucrați cu un calculator modern, cum ar fi un Pentium, încă mai există limitări asupra cantității de memorie pe care o pot accesa programele dumneavoastră. În plus, cu cât mai mare este un obiect, cu atât mai multe prelucrări va necesita fiecare funcție ce manevrează acel obiect. Prin urmare, este important să înțelegeți câtă memorie vor consuma clasele dumneavoastră. Veți calcula cantitatea de memorie necesară unui singur obiect al unei clase în mod similar cu tehnica pe care ați utilizat-o pentru a calcula cantitatea de memorie pe care o necesită o instanță de structură. Calcularea cantității de memorie ce o va consuma o clasă este relativ simplă. Mai întâi trebuie să determinați cantitatea maximă de spațiu pe care o va necesita fiecare instanță a clasei. Petru a înțelege mai bine necesarul de memorie al unei instanțe de clasă, analizați următoarea declarație:

```
class simpla
  {
   public:
     int i, j, k;
     float a, b;
     char c[64];
  }
```

Clasa *simpla* va utiliza 78 de octeți de memorie pentru fiecare instanță: 6 octeți pentru întregi, 8 octeți pentru variabilele în virgulă mobilă și 64 de octeți pentru șirul de caractere. O matrice de clase *simple* care conține 10 elemente va necesita prin urmare 780 de octeți de memorie, cum arătăm în figura 976:



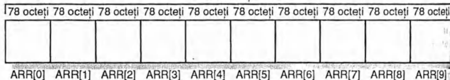


Figura 976 *Model logic al consumului de memorie al unei matrice de clase.*

CODUL DIN INTERIORUL UNEI CLASE POATE FI MODIFICAT

C/C++ 977

După cum ați învățat, C++ vă permite să plasați funcții metode în interiorul clasei sau în afara clasei. Atunci când determinați care funcții să le plasați în interior (inline) sau dacă vreuna dintre funcții va apărea în interior, rețineți că plasarea metodelor în interior expune codul la posibile modificări. De exemplu, următoarea clasă utilizează câteva funcții definite *inline*:

```
class Carte
 {
  public:
   char titlu[256];
   char autor[64];
   float pret;
   void arata_titlu(void) { cout << titlu << '\n'; };
   float da_pret(void) { return(pret); };
   void arata_carte(void);
   void atrib_editura(char *nume) { strcpy(editura, nume); };
  private:
   char editura[256];
   void arata_editura(void) { cout << editura << '\n'; };
 };
```

Atunci când alt programator utilizează clasa de mai sus, el poate schimba cu ușurință metodele clasei, deoarece codul este cuprins în clasa însăși. Dacă, în schimb, plasați metodele clasei într-o bibliotecă de clase, programatorul trebuie să aibă acces la codul sursă al bibliotecii pentru a putea modifica metodele clasei. Utilizând biblioteca de clase, puteți proteja clasa la modificări neașteptate.

STOCAREA DE TIP STATIC

C/C++ 978

După cum ați învățat, documentația de C++ consideră zona *heap* ca *stocare liberă*. Citind articole și cărți despre C++, puteți întâlni și termenul de *stocare statică (static store)*. În cel mai simplu înțeles, *stocarea statică* este o zonă a memoriei globale de la care compilatorul alocă date. Atunci când creați variabile globale sau statice, compilatorul poate aloca memorie pentru variabile din zona de *stocare statică*. În majoritatea cazurilor, domeniul de valabilitate al obiectelor pe care compilatorul le alocă în zona de *stocare statică* este întreg programul. Cu alte cuvinte, obiectul este global.

SINCRONIZAREA OPERAȚIILOR DE I/O UTILIZÂND STDIO

C/C++ 979

Așa cum ați învățat, programele în C++ pot utiliza funcțiile standard de I/O cum ar fi *printf* și *scanf* care sunt definite în *stdio.h*, dar mai pot utiliza și operatorii de inserare și de extragere din fluxurile de I/O *cin* și *cout*. Pentru a face programul dumneavoastră mai ușor de citit, de obicei va trebui să alegeți între o tehnică sau alta. Totuși, uneori nu puteți evita folosirea ambelor metode. În asemenea cazuri, puteți sincroniza operațiile dintre *cout* și *cin* utilizând funcția *sync_with_stdio*. Această funcție indică celor două tehnici de I/O să utilizeze același buffer de intrare și același buffer de ieșire astfel încât aceleași date să fie accesibile ambelor. Următorul program, *syncio.cpp* ilustrează modul de utilizare a funcției *sync_with_stdio*:

```
#include <iostream.h>
#include <stdio.h>

void main(void)
  {
    ios::sync_with_stdio();
    printf("Aceasta carte este: ");
    cout << "Jamsa's C/C++ Programmer's Bible\n";
  }
```

980 FLUXURILE DE I/O DIN C++



Aproape toate secțiunile referitoare la C++ prezentate în cadrul acestei cărți utilizează pe larg fluxurile de I/O *cin*, *cout* și *cerr*. În capitolul despre C++ avansat al acestei cărți, veți învăța despre conceptul de *moștenire*, care permite obiectelor unei clase să moștenească toate caracteristicile unei alte clase. Limbajul C++ pune la dispoziție clasa de bază *ios* (*input-output stream*) care definește operațiile fundamentale de intrare-ieșire. Utilizând fluxul *ios*, limbajul C++ derivează o clasă a fluxului de ieșire și una a fluxului de intrare. Dacă examinați cu atenție fișierul antet *iostream.h* din capitolul de C++ avansat al acestei cărți, veți întâlni o definiție a clasei *ios* precum și definițiile claselor asociate fluxurilor de intrare-ieșire, care vor fi explicate în următoarele secțiuni. Observați că fluxurile sunt definite în cadrul fișierelor antet *iostream.h*, *fstream.h* și *strstream.h*.

981 FLUXURILE DE IEȘIRE DIN C++



În cadrul acestei cărți, programele dumneavoastră au utilizat pe scară largă fluxul de ieșire *cout*. În cel mai simplu sens, un *flux de ieșire* este o destinație a octeților. În explicațiile anterioare, probabil ca ți presupus că limbajul C++ oferă un flux de ieșire utilizat de către *cout*, *cerr* și *clog*, precum și un flux de intrare, utilizat de către *cin*. De fapt, fișierele antet bazate pe fluxuri definesc trei fluxuri de ieșire diferite. Tabelul 981 descrie pe scurt modul de utilizare a fiecăruia dintre aceste trei fluxuri de ieșire.

Flux de ieșire	Funcție
ostream	Utilizat pentru scrierea către *cout*, *cerr* și *clog*.
ofstream	Utilizat pentru scrierea unui fișier pe disc.
ostrstream	Utilizat pentru scrierea dintr-un buffer de ieșire într-un șir de caractere.

Tabelul 981 *Fluxurile de ieșire definite în* ***iostream.h***, ***fstream.h*** *și* ***strstream.h***.

Unele din secțiunile prezentate în cadrul acestui capitol prezintă modalități de utilizare a acestor fluxuri în cadrul programelor dumneavoastră.

982 FLUXURILE DE INTRARE DIN C++



Așa cum ați învățat în secțiunea 981, fișierul antet *iostream.h* definește trei fluxuri de ieșire diferite, unul pentru scrierea pe ecran, altul pentru scrierea într-un fișier și altul pentru scrierea în șiruri de caractere. Așa cum probabil bănuiți, aceste fișiere antet definesc de

asemenea și trei fluxuri de intrare. În cel mai simplu sens, un flux de intrare este o sursă de octeți. Tabelul 982 descrie pe rând funcția fiecăruia dintre aceste trei fluxuri de intrare.

Flux de intrare	Funcție
istream	Utilizat pentru citirea de la *cin*.
ifstream	Utilizat pentru citirea dintr-un fișier de pe disc.
istrstream	Utilizat pentru citirea dintr-un șir de caractere într-un buffer de intrare.

Tabelul 981 *Fluxurile de intrare definite în* ***iostream.h, fstream.h*** *și* ***strstream.h***

Unele din secțiunile ce urmează vor prezenta modalități în care programele dumneavoastră pot utiliza aceste fluxuri.

UTILIZAREA MEMBRILOR CLASEI IOS PENTRU FORMATAREA IEȘIRILOR ȘI INTRĂRILOR

C/C++983

Așa cum ați învățat, puteți utiliza membrii clasei *ios* pentru a formata intrările și ieșirile. De asemenea puteți utiliza indicatoarele de formatare din ios specifice fiecărui tip. Fișierul antet *iostream.h* cuprinde definițiile pentru tipurile specificate de indicatoare anonime.

Indicator	Valoare	Semnificație
skipws	0x0001	Sare peste spațiile albe la intrare
left	0x0002	Aliniere la stânga a ieșirii
right	0x0004	Aliniere la dreapta a ieșirii
internal	0x0008	Adaugă caractere de umplere după orice indicator de bază sau de semn
dec	0x0010	Conversie zecimală
oct	0x0020	Conversie octală
hex	0x0040	Conversie hexazecimală
showbase	0x0080	Utilizarea indicatorului bazei la ieșire
showpoint	0x0100	Forțează scrierea cu virgulă
uppercase	0x0200	Scrierea cu majuscule a numerelor hexazecimale
showpos	0x0400	Adăugarea unui + la întregii pozitivi
scientific	0x0800	Utilizează notația de tip 1.2345E2
fixed	0x1000	Utilizează notația de tip 123.45
unitbuf	0x2000	Curăță toate fluxurile după inserare
stdio	0x4000	curăță *stdin* și *stdout* după inserare
boolalpha	0x8000	Inserează/extrage simbolurile booleene ca text sau ca numere

Tabelul 983 *Tipurile specificate de* ***iosflags*** *(indicatori ios).*

De exemplu, dacă dați valoarea 1 indicatorului *skipws*, la ieșire se vor omite spațiile albe de la început, atunci când programul dumneavoastră îl va utiliza cu un flux. Dacă dați valoarea 1 indicatorului *hex*, veți putea afișa informațiile la ieșire în reprezentare hexazecimală.

984 INDICATOARE DE FORMATARE



Există câteva modalități pentru a utiliza indicatoarele de formatare într-un flux. Totuși, cel mai simplu și prin urmare și cel mai utilizat mod este de a apela funcția membru *setf*, pe care programele dumneavoastră o vor implementa ca mai jos:

```
#include <iostream.h>
long setf(long flags);
```

Dacă apelați funcția cu un parametru gol, ea va returna valoarea precedentă a indicatorului de format. Dacă o apelați cu unul dintre parametrii specificați, ea va da valoarea indicatorului specificat de parametru. De exemplu, pentru a da valoarea 1 indicatorului *showpos*, veți utiliza următorul apel de funcție:

```
flux.setf(ios::showpos);
```

Numele *flux* corespunde fluxului către care se va efectua ieșirea – de exemplu *cout*, *cerr* și așa mai departe. Pentru a înțelege mai bine această procesare, studiați următorul program, *show_hex.cpp*, care afișează o valoare în format hexazecimal:

```
#include <iostream.h>
void main(void)
  {
    cout.setf(ios::hex);
    cout.setf(ios::showbase);
    cout << 100;
  }
```

985 ȘTERGEREA INDICATOARELOR DE FORMAT



Așa cum ați învățat, puteți utiliza funcția *setf* pentru a da valoarea 1 indicatoarelor de formatare *ios*. Funcția *unsetf* șterge indicatoarele pe care le-ați stabilit anterior cu funcția *setf*. Ca și *setf*, *unsetf* este ușor de implementat, așa cum arătăm mai jos:

```
#include <iostream.h>
long unsetf(long flags);
```

De exemplu, să presupunem că doriți să poziționați pe 1 un indicator într-un program care să forțeze calculatorul să genereze întotdeauna ieșirile în notație științifică. Totuși, anumite funcții din programul dumneavoastră probabil că vor dori să dezactiveze indicatorul înainte de a genera ieșirea. Următorul program, *unsetf.cpp*, poziționează și apoi șterge indicatorul *uppercase*:

```
#include <iostream.h>
void main(void)
  {
    cout.setf(ios::uppercase | ios::scientific);
    cout << 100.12;
```

```
    cout.unsetf(ios::uppercase);
    cout << endl << 100.12;
  }
```

UTILIZAREA FUNCȚIEI SETF SUPRAÎNCĂRCATE

C/C++ 986

În secțiunile precedente ați utilizat funcția *setf* pentru a controla ieșirile pe ecran. Pe lângă implementarea cu un singur parametru a funcției *setf*, *iostream.h* mai pune la dispoziție și o versiune supraîncărcată a funcției, pe care o veți implementa în programele dumneavoastră așa cum este arătat mai jos:

```
#include <iostream.h>
long setf(long flags1, long flags2);
```

Atunci când utilizați versiunea supraîncărcată a funcției *setf*, programul dumneavoastră va anula indicatoarele specificate în parametrul *flags2* și apoi va poziționa indicatoarele specificate în parametrul *flags1*. De exemplu, următorul program, *setf_supra.cpp*, șterge indicatoarele *showpos* și *showpoint* și poziționează din nou indicatorul *showpoint*:

```
#include <iostream.h>
void main(void)
  {
    cout.setf(ios::showpos | ios::showpoint);
    cout << 100 << endl;
    cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
    cout << 100;
  }
```

EXAMINAREA INDICATORELOR DE FORMAT CURENTE

C/C++ 987

Pe măsură ce continuați să lucrați cu indicatoare de format, puteți să cunoașteți doar care sunt valorile curente ale formatului, fără a le mai schimba. Pentru a vă ajuta să determinați valorile curente de format, clasa *ios* pune la dispoziție funcția membru flags, care returnează valorile curente ale fiecărui indicator, codificate ca întreg de tip *long*. Următorul program, *afisflag.cpp*, utilizează o funcție definită de utilizator denumită *afisflags*, care împreună cu funcția *ios flags* generează informații despre valorile curente ale sistemului:

```
#include <iostream.h>
void afisflags(void);
void main(void)
  {
    afisflags();
    cout.setf(ios::right | ios::showpoint | ios::fixed);
```

```
    afisflags();
  }
void afisflags(void)
  {
    long flag_set, i;
    int j;
    char flags[15][12] = {
     "skipws", "left", "right", "internal", "dec",
     "oct", "hex", "showbase", "showpoint", "uppercase",
     "showpos", "scientific", "fixed", "unitbuf",
    };
    flag_set = cout.flags();
    for (i=1, j=0; i<0x2000; i = i<<1, j++)
     if (i & flag_set)
       cout << flags[j] << " este activat." << endl;
     else
       cout << flags[j] << " este dezactivat." << endl;
    cout << endl;
  }
```

Atunci când compilați și executați programul *afisflag.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
skipws este activat.
left este dezactivat.
right este dezactivat.
internal este dezactivat.
dec este dezactivat.
oct este dezactivat.
hex este dezactivat.
showbase este dezactivat.
showpoint este dezactivat.
uppercase este dezactivat.
showpos este dezactivat.
scientific este dezactivat.
fixed este dezactivat.

skipws este activat.
left este dezactivat.
right este activat.
internal este dezactivat.
dec este dezactivat.
oct este dezactivat.
hex este dezactivat.
showbase este dezactivat.
showpoint este activat.
uppercase este dezactivat.
showpos este dezactivat.
scientific este dezactivat.
fixed este activat.
C:/>
```

POZIȚIONAREA TUTUROR INDICATOARELOR

C/C++988

Așa cum clasa *ios* supraîncarcă funcția *setf*, ea supraîncarcă și funcția *flags*. Funcția *flags* supraîncărcată vă permite să activați toate indicatoarele de format asociate cu un anumit flux. Cu alte cuvinte, programul dumneavoastră poate crea o mască de indicatoare, apoi să apeleze funcția *flags* și să activeze toate indicatoarele detaliate în cadrul măștii. De exemplu, următorul program activează indicatoarele *showpos*, *showbase*, *oct* și *right* utilizând funcția *flags*. De asemenea, afișează indicatoarele înainte și după schimbarea valorilor. Funcția *afisflags* este aceeași ca funcția apelată în programul *afisflag.cpp* din secțiunea 987 (care nu este retipărită din motive de spațiu, dar este inclusă în fișierul sursă al programului *set_allf.cpp* de pe CD-ROM-ul acestei cărți). Funcția *main* a programului *set_allf.cpp* este următoarea:

```
void main(void)
  {
    afisflags();
    long f = ios::showpos | ios::showbase | ios::oct | ios::right;
    cout.flags(f);
    afisflags();
  }
```

UTILIZAREA FUNCȚIEI PRECISION

C/C++989

În secțiunea 830 ați utilizat funcția *setprecision* pentru a controla precizia cu care programele dumneavoastră afișează datele în virgulă mobilă. De asemenea, puteți controla precizia cu care *cout* afișează datele în virgulă mobilă cu ajutorul funcției membru *ios precision*. Veți implementa funcția membru *precision* ca mai jos:

```
#include <iostream.h>

int precision(int p);
```

Atunci când programul dumneavoastră apelează funcția *precision*, el va stabili noua precizie la numărul *p* de poziții zecimale specificate și va returna către funcția apelantă valoarea precedentă a preciziei. Valoarea implicită a preciziei pe care o utilizează *cout* este de șase poziții. În funcție de versiunea compilatorului dumneavoastră, s-ar putea să fie nevoie ca valoarea pentru precizie să fie stabilită înainte de fiecare apelare pentru cout, altfel cout va folosi valorile implicite. Următorul program, *ios_prec.cpp* utilizează funcția *precision* pentru a formata ieșirea:

```
#include <iostream.h>

void main(void)
  {
    int i;
    float valoare = 1.2345;
    for (i = 0; i < 4; i++)
      {
```

```
        cout.precision(i);
        cout << valoare << endl;
      }
   }
```

990 UTILIZAREA FUNCȚIEI FILL

C/C++

Așa cum ați învățat în secțiunea 828, programele dumneavoastră pot utiliza funcția membru *fill* pentru a schimba caracterul pe care fluxurile dumneavoastră de ieșire îl utilizează pentru a umple spațiile goale (implicit se va folosi caracterul spațiu). Pe măsură ce programele dumneavoastră devin mai complexe, utilizarea funcției membru *fill* vă poate ajuta în a face ieșirile programelor dumneavoastră mai folositoare și mai ușor de citit pentru utilizator. De exemplu, următorul program, *bun_tex.cpp*, utilizează funcțiile membru *width*, *precision* și *fill* pentru a genera ieșirea formatată:

```
#include <iostream.h>

void main(void)
  {
    cout.precision(4);
    cout.width(10);
    cout << 10.12345 << endl;
    cout.width(10);
    cout.fill('-');
    cout << 10.12345 << endl;
    cout.width(10);
    cout << "Hi!" << endl;
    cout.width(10);
    cout.setf(ios::left);
    cout << 10.12345;
  }
```

Atunci când compilați și executați programul *bun_tex.cpp*, el va genera următorul rezultat:

```
     10.12
----10.12
------Hi!
10.12----
C:\>
```

991 MANIPULATORII

C/C++

În secțiunile precedente ați învățat că puteți utiliza comenzi din cadrul fluxului de ieșire *cout* pentru a controla afișarea textului. Aceste comenzi sunt cunoscute sub numele de *manipulatori*. Manipulatorii sunt folositori deoarece vă permit să formatați textul cu ajutorul unui număr minim de comenzi. De exemplu, pentru a stabili lărgimea afișării la valoarea 10 și caracterul de umplere la '*', programele dumneavoastră pot executa următoarele comenzi:

```
cout.width(10);
cout.fill('*');
cout<< "Exemplu"<<endl;
```

Ca o alternativă, programele dumneavoastră pot utiliza manipulatori pentru a obține același rezultat, în mai puține linii de program, ca mai jos:

```
cout << setw(10) << setfill('*') << "Exemplu"<< endl;
```

UTILIZAREA MANIPULATORILOR PENTRU A FORMATA INTRĂRILE ȘI IEȘIRILE

C/C++ 992

Așa cum ați învățat, programele dumneavoastră pot utiliza manipulatori în locul funcțiilor membre ale fluxurilor pentru formatarea intrărilor și a ieșirilor. Fișierele antet *iostream.h* și *iomanip.h* definesc manipulatorii. Fișierul antet *iomanip.h* definește numai manipulatorii care primesc parametrii (cum este *setw*). Tabelul 992 arată care sunt manipulatorii pe care îi puteți utiliza în programele dumneavoastră, scopul lor și dacă îi puteți utiliza pentru intrări și/sau ieșiri:

Modelator	Intrare/Ieșire	Scop
dec	Intrare/Ieșire	Indică fluxului să afișeze/citească datele în format zecimal
endl	Ieșire	Indică fluxului să afișeze un caracter de linie nouă și să se șteargă
ends	Ieșire	Indică fluxului să afișeze un NULL
flush	Ieșire	Indică fluxului să se șteargă
hex	Intrare/Ieșire	Indică fluxului să afișeze/citească datele în format hexazecimal
oct	Intrare/Ieșire	Indică fluxului să afișeze/citească datele în format octal.
resetiosflasg(long f)	Intrare/Ieșire	Dezactivează indicatoarele *f* specificate
setbase(int baza)	Ieșire	Stabilește baza de numerație la valoarea *baza*
setfill(int ch)	Ieșire	Stabilește caracterul de umplere la *ch*
setiosflags(long f)	Intrare/Ieșire	Activează indicatoarele *f* specificate
setprecision(int p)	Ieșire	Stabilește numărul de cifre de precizie
setw(int w)	Ieșire	Stabilește lățimea câmpului la *w*
ws	Intrare	Sare peste spațiile albe de la început

Tabelul 992 *Manipulatorii de flux pe care îi puteți folosi pentru a formata intrările și ieșirile.*

O COMPARAȚIE ÎNTRE MANIPULATORI ȘI FUNCȚIILE MEMBRE

C/C++ 993

Așa cum ați văzut, programele dumneavoastră pot utiliza manipulatori, funcții membre sau pe amândouă. De exemplu, următorul program, *cu_manip.cpp* utilizează un manipulator pentru a controla fluxul de ieșire:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    cout << hex << 100 << endl;
    cout << setfill('?') << setw(10) << 2343.10 << endl;
  }
```

Deși programul *cu_manip.cpp* se va compila ceva mai lent (deoarece include *iomanip.h*), codul său sursă este mult mai compact decât cel al programului *cu_mem.cpp* arătat mai jos:

```
#include <iostream.h>

void main(void)
  {
    cout.setf(ios::hex);
    cout << 100 << endl;
    cout.unsetf(ios::hex);
    cout.fill('?');
    cout.width(10);
    cout << 2343.10 << endl;
  }
```

Atunci când compilați și executați programele *cu_manip.cpp* și *cu_mem.cpp*, amândouă vor fi de dimensiuni egale. Ambele programe se vor executa în timp egal și sunt echivalente din punctul de vedere al modului în care afectează programul. Este bine să utilizați metoda care vi se va părea mai clară și să o luați drept standard.

994 CREAREA PROPRIILOR FUNCȚII DE INSERARE C/C++

În multe din secțiunile precedente ați utilizat funcția de inserare (operatorul <<) pentru a face o ieșire la un flux. Pe măsură ce clasele dumneavoastră vor deveni mai complexe, puteți să suprascrieți funcția de inserare. Atunci când suprascrieți funcția de inserare, operatorul de inserare va trebui să returneze o referință la un flux de I/O. Toate funcțiile de inserare supraîncărcate pe care le veți crea vor fi de următorul format general:

```
ostream& operator<<(ostream &flux, tip_clasa obiect)
  {
    // corpul functiei de inserare
    return flux;
  }
```

Funcția returnează o referință de tipul *ostream* – o clasă pe care C++ o derivează din clasa *ios* și care acceptă ieșirea. Primul parametru al operatorului este fluxul în care se va insera, iar al doilea este obiectul care să fie afișat în flux.

În general, funcțiile dumneavoastră de inserare vor face aproximativ aceiași pași pe care îi fac de obicei, numai că îi vor face prin fluxul obiect și nu direct prin *cout*, *cerr* sau *clog*. De obicei veți formata și veți structura afișarea datelor membre ale unei clase în cadrul unei funcții supraîncărcate, așa cum veți vedea în secțiunea 996.

SUPRAÎNCĂRCAREA OPERATORULUI DE EXTRAGERE

C/C++995

În secțiunile precedente, ați supraîncărcat câțiva operatori în relație cu anumite clase. Atunci când clasele dumneavoastră conțin anumiți membri a căror valoare doriți să o afișați într-un anumit format, utilizarea lui *cout* și a operatorului de *extragere* poate conduce la generarea unei cantități considerabile de cod scris. De exemplu, să presupunem că lucrați cu o clasă ai cărei membri includ un nume, sexul (M sau F), vârsta și numărul de telefon, pe care doriți să le afișați după cum urmează:

```
Nume: John Doe      Sexul: M      Varsta: 43      Telefon: 555-1212
```

Utilizând *cout*, programele dumneavoastră vor trebui să cuprindă următoarea instrucțiune de fiecare dată când vor dori să afișeze datele:

```
cout << "Nume: " << nume << "\tSexul: " << sex << "\t Varsta: "
     << varsta << "\tTelefon: " << telefon << endl;
```

O alternativă mai bună ar fi supraîncărcarea operatorului de extragere, așa cum este arătat în următorul program, *outsupr.cpp*:

```
#include <iostream.h>
#include <string.h>

class Angajat
  {
    public:
      Angajat(char *nume, char sex, int varsta, char *telefon)
        {
          strcpy(Angajat::nume, nume);
          Angajat::sex = sex;
          Angajat::varsta = varsta;
          strcpy(Angajat::telefon, telefon);
        };
      friend ostream& operator<< (ostream& cout, Angajat ang);
    private:
      char nume[256];
      char telefon[64];
      int varsta;
      char sex;
  };
ostream& operator<< (ostream& cout, Angajat ang)
  {
    cout << "Nume: " << ang.nume << "\tSex: " << ang.sex;
    cout << "\tVarsta: " << ang.varsta << "\tTelefon: " <<
          ang.telefon << endl;
    return cout;
  }
void main(void)
  {
```

```
    Angajat muncitor("Happy", 'M', 4, "555-1212");
    cout << muncitor ;
  }
```

Programul *outsupr.cpp* supraîncarcă operatorul de extragere din clasa *ostream*. Programul va utiliza doar operatorul de extragere supraîncărcat atunci când programul apelează operatorul cu clasa *Angajat*. Prin urmare, C++ nu apelează funcția supraîncărcată atunci când efectuează operații de ieșire în cadrul supraîncărcării înseși, ci va lucra cu extractorul obișnuit al fluxului de ieșire. Deoarece operatorul de extragere trebuie să acceseze datele membre ale clasei *Angajat*, programul va declara acest operator ca operator *friend* al clasei.

996 ALTĂ MODALITATE DE A SUPRAÎNCĂRCA OPERATORUL DE INSERARE PENTRU COUT

C/C++

Majoritatea programelor prezentate pe parcursul ultimelor două capitole ale acestei cărți au utilizat în mare măsură fluxul *cout* pentru a afișa ieșirea pe ecran. Următorul program, *cout_maj.cpp*, supraîncarcă operatorul de inserare pentru *cout* pentru șiruri de caractere, indicându-i să afișeze întotdeauna șirurile de caractere cu majuscule:

```
#include <iostream.h>
#include <string.h>

ostream& operator<<(ostream& cout, char *sir)
  {
    char *sr = strupr(sir);
    while (*sr)
      cout.put(*sr++);
    return(cout);
  }
void main(void)
  {
    cout << "Acesta este un test";
    cout << "\nJamsa's C/C++ Programmer's Bible";
  }
```

În cadrul funcției supraîncărcate înseși, instrucțiunile utilizează funcția *strupr* pentru a converti șirurile de caractere în majuscule și apoi utilizează cout pentru a afișa câte un caracter o dată. Funcția supraîncărcată nu va putea utiliza operatorul de inserare din *cout* pentru a afișa șirul de caractere, pentru că se va genera o serie infinită de apeluri recursive.

997 CREAREA PROPRIILOR FUNCȚII DE EXTRAGERE

C/C++

Tot așa cum puteți crea propriile dumneavoastră funcții de inserare, tot așa puteți supraîncărca operatorul de extragere (>>) în clasele dumneavoastră. În general, veți supraîncărca operatorul de extragere pentru a obține un control mai bun asupra intrărilor de la utilizator către datele membre ce alcătuiesc o clasă. Veți supraîncărca funcția extractor utilizând următoarea formă generală:

```
istream &operator>>(istream &flux, tip_clasa obiect)
  {
    // corpul functiei de extragere
    return flux;
  }
```

Observați că, spre deosebire de funcțiile supraîncărcate de inserare, trebuie să transmiteți o referință către obiectul din cadrul funcției de extragere supraîncărcate. Funcțiile de extragere returnează o referință la un flux de tipul *istream*, care este derivat din clasa flux de intrare *ios*. *Primul parametru este o referință la un flux (în general cin).*

UN EXEMPLU DE EXTRACTOR

C/C++ 998

În secțiunea 997 ați învățat formatul de baza pentru crearea unei funcții extractor supraîncărcate. Atunci când creați funcții extractor globale supraîncărcate, veți crea o funcție extractor supraîncărcată specifică pentru fiecare clasă. De exemplu, următorul program, *inover.cpf* utilizează clasa *Angajat* din secțiunea 995 și primește intrări special formatate pentru această clasă:

```
#include <iostream.h>
#include <string.h>
class Angajat
  {
   public:
    Angajat(void) {};
    Angajat(char *nume, char sex, int varsta, char *telefon)
      {
        strcpy(Angajat::nume, nume);
        Angajat::sex = sex;
        Angajat::varsta = varsta;
        strcpy(Angajat::telefon, telefon);
      };
    friend ostream &operator<<(ostream &cout, Angajat ang);
    friend istream &operator>>(istream &flux, Angajat &ang);
   private:
    char nume[256];
    char telefon[64];
    int varsta;
    char sex;
  };
ostream &operator<<(ostream &cout, Angajat ang)
  {
   cout << "Nume: " << ang.nume << "\tSex: " << ang.sex;
   cout << "\tVarsta: " << ang.varsta << "\tTelefon: "
     << ang.telefon << endl;
   return cout;
  }
```

```
istream &operator>>(istream &flux, Angajat &ang)
  {
    cout << "Scrieti numele: ";
    flux >> ang.nume;
    cout << "Scrieti sexul: ";
    flux >> ang.sex;
    cout << "Scrieti varsta: ";
    flux >> ang.varsta;
    cout << "Scrieti numarul de telefon: ";
    flux >> ang.telefon;
    return flux;
  }
void main(void)
  {
    Angajat muncitor;
    cin >> muncitor;
    cout << muncitor;
  }
```

999 CREAREA PROPRIILOR FUNCȚII MANIPULATOR C/C++

Programele dumneavoastră, pe lângă supraîncărcarea operatorilor de inserare și de extragere, pot și să creeze propriile funcții manipulator. Crearea de manipulatori personalizați poate fi folositoare din două motive. Mai întâi, dumneavoastră puteți reuni o secvență de câteva funcții distincte de intrare-ieșire într-un singur apel de funcție membră, făcând codul programului mai ușor de folosit și de înțeles. În al doilea rând, puteți crea manipulatori personalizați care să vă ajute să manevrați intrările și ieșirile către/dinspre dispozitive nestandard. În general, vă veți construi manipulatorii personalizați astfel:

```
nume-flux &nume-manipulator(nume-flux &flux [, parametri])
  {
    // cod specific manipulatorului
    return flux;
  }
```

În următoarele două secțiuni veți crea câțiva manipulatori personalizați pentru ieșiri. Veți putea aplica tehnicile folosite în aceste secțiuni la oricare din funcțiile manipulator pe care le veți crea.

1000 CREAREA MANIPULATORILOR FĂRĂ PARAMETRI C/C++

După cum ați învățat în secțiunea 999, puteți crea manipulatori personalizați în cadrul unui program. Puteți crea fie manipulatori parametrizați, fie manipulatori fără parametri. Secțiunea 1001 se va ocupa de manipulatorii parametrizați. Oricum, dacă manipulatorul dumneavoastră realizează o activitate standard, care nu va necesita intrări de la instrucțiunea apelantă, veți utiliza manipulatori fără parametri. Următorul program, *setbex.cpp*, creează și utilizează un manipulator fără parametri:

```
#include <iostream.h>
#include <iomanip.h>
ostream &sethex(ostream &flux)
  {
    flux.setf(ios::showbase);
    flux.setf(ios::hex);
    return flux;
  }
void main(void)
  {
    cout << 256 << " " << sethex << 256;
  }
```

UTILIZAREA PARAMETRILOR CU MANIPULATORII

C/C++1001

În secțiunea 1000 ați învățat cum să creați o funcție manipulator fără parametri. După cum ați învățat, crearea unui manipulator fără parametri este relativ simplă. Din păcate, crearea unui manipulator care acceptă unul sau mai multe argumente, cum este *setw(5)*, nu mai este așa de simplă. Pentru a crea un manipulator care acceptă unul sau mai mulți parametri, trebuie să creați manipulatorul utilizând o *clasă generică*. Așa cum veți învăța în secțiunile care vor urma, clasele generice vă permit să scrieți clase și funcții care acceptă tipuri multiple, fără a mai supraîncărca clasa sau funcția pentru fiecare tip în parte. Veți învăța mai multe despre clasele generice în secțiunea 1124. Deocamdată vă propunem totuși să studiați următoarea construcție a unui manipulator tipic cu parametri:

```
ostream &indent(ostream &flux, int lung)
  {
    register int i;
    for(i = 0; i < lung; i++)
      cout << " ";
    return flux;
  }
```

Atunci când apelați manipulatorul parametrizat *indent*, valoarea pe care o veți trece manipulatorului va stabili numărul de spații cu care programul va indenta fluxul.

VECHEA BIBLIOTECĂ DE CLASE DE FLUXURI

C/C++1002

După cum ați învățat, multe programe în C++ vor utiliza fișierul antet *iostream.h*, care conține definițiile a multe dintre controalele de intrare/ieșire ale C++. Atunci când Bjarne Stroustrup a inventat limbajul C++, el utiliza o bibliotecă de clase de intrare/ieșire mai mică și oarecum diferită, denumită *stream.h*. Pe măsură ce limbajul C++ a evoluat, vechea bibliotecă *stream.h* a fost înlocuită de biblioteca *iostream.h*, descrisă în această carte. Pentru a menține compatibilitatea cu programele mai vechi, majoritatea compilatoarelor de C++ continuă să accepte biblioteca *stream.h*. Totuși, în programele dumneavoastră va trebui să utilizați întotdeauna biblioteca mai nouă și mai bine dotată, *iostream.h*.

1003 DESCHIDEREA UNUI FIȘIER FLUX

C/C++

După cum ați învățat, C++ pune la dispoziție fluxurile fișier *ifstream* și *ofstream* (de intrare, respectiv ieșire). În programele dumneavoastră în C++, puteți efectua intrări și ieșiri cu ajutorul acestor două clase sau mai puteți utiliza operațiile de intrare-ieșire ale limbajului C standard utilizând *fopen*, *fgets*, *fputs* și așa mai departe. Pentru a deschide un flux, trebuie să declarați o variabilă de clasă corespunzătoare (*ifstream* sau *ofstream*). Presupunând că doriți să efectuați operații de intrare și de ieșire, declarațiile dumneavoastră vor arăta ca mai jos:

```
ifstream intrare;
ofstream iesire;
```

Prin urmare, pentru a deschide un flux, va trebui să utilizați funcția membru *open*, ca mai jos:

```
intrare.open("NUMEFISIER.EXT", ios::in);
iesire.open("NUMEFISIER.OUT", ios::out);
```

Forma generală a instrucțiunii *open* este:

```
ifstream.open(const char *NUMEFISIER, int nMod=ios::in, int
              nProt=filebuf::openprot);
ofstream.open(const char *NUMEFISIER, int nMod=ios::out, int
              nProt=filebuf::openprot);
```

După cum puteți vedea, valoarea implicită a parametrului *nMod* este *ios::in* sau *ios::out*, în funcție de fluxul manipulat, astfel încât următoarele construcții să fie la fel de valide ca și cele arătate mai sus:

```
intrare.open("NUMEFISIER.EXT");
iesire.open("NUMEFISIER.OUT");
```

Parametrul *nProt* vă permite controlarea accesului partajat la fișier. În plus față de utilizarea membrului *open*, mai puteți utiliza și funcția constructor a obiectului flux atunci când declarați variabila flux, ca mai jos:

```
ifstream intrare("NUMEFISIER.EXT", ios::in);
ofstream iesire("NUMEFISIER.OUT", ios::out);
```

Parametrii *ios::in* sau *ios::out* fac distincția între intrare sau ieșire. În afară de aceste valori, programele dumneavoastră mai pot utiliza combinații ale valorilor listate în tabelul 1003.

Valoare	Semnificație
ios::app	Deschide fluxul în mod adăugare
ios::ate	Deschide un fișier fie pentru intrare, fie pentru ieșire, mutând pointerul de fișier la sfârșitul lui
ios::in	Deschide un fișier pentru intrare
ios::out	Deschide un fișier pentru ieșire
ios::nocreate	Deschide un fișier numai dacă el există deja
ios::noreplace	Deschide un fișier numai dacă el nu a fost creat deja

Valoare	Semnificație
ios::trunc	Trunchiază un fișier existent
ios::binary	Deschide un fișier în mod binar

Tabelul 1003 *Valorile modului de deschidere pentru **ifstream** și **ofstream**.*

ÎNCHIDEREA UNUI FIȘIER FLUX

C/C++1004

După cum ați învățat în secțiunea 1003, programele dumneavoastră pot deschide un flux *iostream* utilizând fie membrul *open*, fie o funcție constructor atunci când declarați o variabilă flux. Atunci când terminați de utilizat fișierul flux, programele ar trebui să închidă fluxul, utilizând funcția membru *close*, așa cum arătăm mai jos:

```
intrare.close();
iesire.close();
```

În mod implicit, atunci când programul dumneavoastră își încheie execuția sau dacă distrugeți variabila clasă, C++ închide fișierul flux. Dacă doriți să asociați fișierul flux cu un alt fișier, mai întâi trebuie să folosiți membrul *close*, apoi să folosiți noul nume de fișier pentru a redeschide fluxul.

CITIREA ȘI SCRIEREA DATELOR ÎN FLUXURILE FIȘIERE

C/C++1005

După cum ați învățat, limbajul C++ permite programelor dumneavoastră să deschidă fluxuri fișiere ale clasei *istream* pentru operații de intrare. Pentru a citi date dintr-un flux fișier de intrare, programele dumneavoastră folosesc funcția membru *read*, arătată mai jos:

```
intrare.read(buffer, nr_de_octeti);
```

După cum puteți vedea, programul trebuie să precizeze un buffer în care funcția membru *read* să depoziteze datele citite, iar programul trebuie să informeze funcția membru *read* despre numărul de octeți pe care dorește să-i citească. În mod similar, dacă ați deschis un flux de ieșire, programele dumneavoastră pot folosi funcția membru *write* pentru a scrie date în fluxul de ieșire, ca mai jos:

```
iesire.write(buffer, nr_de_octeti);
```

Observați că membrii *read* și *write* nu returnează numărul de octeți pe care funcțiile l-au scris sau citit cu succes. În schimb, funcțiile returnează referințe la flux. Pentru a determina succesul sau eșecul unei operații, trebuie să testați membrii de stare, detaliați în secțiunea 1006. Operațiile pot eșua din diverse motive, incluzând: discul plin, fișiere inexistente, defecte ale discului și altele. Trebuie să vă asigurați că programele dumneavoastră testează în mod regulat succesul sau eșecul unei activități de I/O pentru a vă proteja împotriva erorilor necunoscute.

1006 TESTAREA STĂRII UNEI OPERAȚII CU FIȘIERE C/C++

Atunci când efectuați operații de I/O pe fișiere cu ajutorul claselor de fluxuri *ifstream* și *ofstream*, programele dumneavoastră pot utiliza membrii listați în tabelul 1006 pentru a determina dacă o operație de deschidere, citire sau scriere a reușit.

Membru	Exemplu	Funcție
bad	*stream.bad()*	Returnează valoarea adevărat dacă operația de I/O întâlnește o eroare ireparabilă
fail	*stream.fail()*	Returnează valoarea adevărat dacă operația de I/O întâlnește o eroare reparabilă sau previzibilă, cum ar fi un fișier care nu a fost găsit
good	*stream.good()*	Returnează valoarea adevărat dacă operația de I/O a reușit
eof	*stream.eof()*	Returnează valoarea adevărat dacă operația de I/O întâlnește sfârșitul de fișier
clear	*stream.clear()*	Șterge indicatoarele de stare
rdstate	*stream.rdstate()*	Returnează starea curentă de eroare

***Tabelul 1006** Funcțiile membre care returnează informații despre succesul sau eșecul unei operații de I/O.*

1007 UTILIZAREA MAI MULTOR OPERAȚII CU FIȘIERELE FLUX C/C++

Câteva din secțiunile precedente au prezentat fișierele *flux* ale limbajului C++ și diversele funcții membre pe care programele dumneavoastră le pot utiliza pentru a efectua operații de intrare-ieșire. Următorul program, *copfis.cpp* utilizează câteva dintre aceste funcții membre pentru a crea un program simplu de copiere de fișiere, care copiază fișiere text citind câte un caracter o dată:

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>

void main(int argc, char **argv)
  {
    char buffer[1];
    ifstream input(argv[1], ios::in);
    if (input.fail())
      {
        cout << "Eroare la deschiderea fisierului " << argv[1];
        exit(1);
      }
    ofstream output(argv[2], ios::out);
    if (output.fail())
      {
        cout << " Eroare la deschiderea fisierului " << argv[2];
```

```
        exit(1);
      }
    do {
      input.read(buffer, sizeof(buffer));
      if (input.good())
        output.write(buffer, sizeof(buffer));
    } while (! input.eof());
    input.close();
    output.close();
  }
```

Programul *copfis.cpp* deschide mai întâi fișierul specificat în linia de comandă ca fișier ce trebuie copiat. Dacă acțiunea de deschidere a fișierului eșuează, programul se termină cu o valoare de stare de eșec. Dacă programul a deschis cu succes primul fișier, deschide apoi și fișierul destinație specificat în al doilea argument al liniei de comandă. Dacă programul nu poate deschide al doilea fișier, el se va termina de asemenea cu stare de eroare. Dacă ambele fișiere au fost deschise cu succes, programul va copia primul fișier, câte un caracter o dată, în cel de-al doilea fișier. După ce bucla care copiază caractere ajunge la capătul primului fișier și, prin urmare, se încheie, programul „face curățenie în urma sa" (adică închide fișierele flux deschise) și se sfârșește în mod normal.

Pentru a utiliza programul *copfis.cpp*, trebuie să apelați programul *copfis* așa cum arătam mai jos, pentru a copia fișierul *copfis.cpp* în *copfis.sav*:

```
C:\> COPFIS COPFIS.CPP COPFIS.SAV <ENTER>
```

EFECTUAREA UNEI OPERAȚII DE COPIERE BINARĂ

C/C++ 1008

În secțiunea 1007 ați creat programul *copfis.cpp* care copiază primul fișier text specificat în linia de comandă într-un fișier al cărui nume este specificat în cel de-al doilea argument al liniei de comandă. Dacă doriți să copiați fișiere binare (cum ar fi un fișier EXE care conține un program), trebuie să schimbați operațiile de deschidere ca să utilizeze indicatorul *ios::binary*, ca mai jos:

```
  ifstream intrare(argv[1], ios::in | ios::binary);
  if (intrare.fail())
 {
  cout << "Eroare la deschiderea fisierului " << argv[1];
  exit(1);
 }
  ofstream iesire(argv[2], ios::out | ios::binary);
  if (iesire.fail())
 {
  cout << "Eroare la deschiderea fisierului " << argv[2];
  exit(1);
 }
```

CD-ROM-ul care însoțește această care conține programul *bin_copy.cpp* care efectuează o copiere de fișiere binare.

1009 CLASA STREAMBUF



În secțiunile precedente, ați lucrat cu diverse fișiere și alte fluxuri de intrare/ieșire pentru a manipula date. Atunci când utilizați buffere de date în programele dumneavoastră, aceasta înseamnă că plasați aceste date într-o locație intermediară înainte sau după efectuarea unei operații de citire/scriere. C++ derivă clasele care stochează în buffer date de intrare/ieșire (stochează în buffer fluxuri de I/O) din clasa de bază *streambuf*. Un flux de I/O prin buffer pune la dispoziție o interfață de tip buffer între datele dumneavoastră și zonele de stocare, cum ar fi memoria sau dispozitive fizice. Bufferele pe care obiectele de tip *streambuf* le creează sunt cunoscute ca zone *get*, *put* și *reserve*. Programele dumneavoastră accesează și manevrează conținutul zonelor create cu obiectele *streambuf* cu ajutorul pointerilor care indică spre caracterele din aceste zone *get*, *put* și *reserve*.

Acțiunile de stocare în buffere efectuate de obiectele *streambuf* sunt relativ primitive. Deoarece stocarea în buffere efectuată de obiectele *streambuf* nu este la fel de utilă ca cea pe care o pot efectua clasele flux de nivel înalt derivate de C++ din *streambuf*, aplicațiile dumneavoastră vor obține acces la buffere și funcții de stocare în buffere prin intermediul unui pointer la *streambuf*. Programele dumneavoastră vor utiliza în mod indirect acest pointer, în cadrul definiției unui obiect bazat pe *ios*. Clasa *ios* prevede un pointer la clasa *streambuf* care îi furnizează acces la serviciile de stocare în buffere pentru clase de nivel înalt – cu alte cuvinte, programele dumneavoastră trebuie să utilizeze clasa *ios* și să lase bufferul *streambuf* de intrare/ieșire „în fundal", să lucreze singur. Clasele de nivel înalt pun la dispoziție formatarea intrărilor și ieșirilor. Cu alte cuvinte, programele dumneavoastră ar trebui în general să utilizeze clasele de fluxuri de nivel înalt pentru a controla manipularea intrărilor și ieșirilor. Totuși, probabil că uneori programele dumneavoastră vor trebui să efectueze accesări de nivel jos pe un obiect *streambuf*.

Secțiunea 1010 arată cum să accesați obiectele *streambuf* dintr-un program. Tabelul 1009 listează câteva dintre funcțiile membre publice ale clasei *streambuf* și descrierea lor.

Funcție	Descriere
in_avail	Funcția membru *in_avail* returnează numărul de caractere rămase în bufferul de intrare intern
out_waiting	Funcția membru *out_waiting* returnează numărul de caractere rămase în bufferul de ieșire intern
pbump(n)	Funcția membru *pbump* incrementează pointerul *put* (*pptr*) cu *n*, care poate fi o valoare pozitivă sau negativă
sbumpc	Funcția membru *sbumpc* returnează caracterul curent din bufferul intern de intrare, apoi avansează pointerul intern al bufferului la următorul caracter
sgetc	Funcția membru *sgetc* preia următorul caracter din bufferul intern de intrare
snextc	Funcția membru *snextc* avansează pointerul intern al bufferului de intrare la următorul caracter și returnează acest caracter
sputbackc	Funcția membru *sputbackc* returnează un caracter către bufferul intern de intrare
sputc	Funcția membru *sputc* pune un caracter în bufferul intern de ieșire
stossc	Funcția membru *stossc* avansează pointerul intern al bufferului de intrare la următorul caracter din bufferul de intrare

***Tabelul 1009** Funcțiile membre ale clasei **streambuf**.*

Dacă observați cu atenție descrierile funcțiilor membre, veți vedea ca majoritatea lor efectuează activități similare cu operațiile de nivel înalt pentru care utilizați *get, put, read* și *write*. De fapt, toate clasele bazate pe fluxuri derivă din *streambuf*, toate funcțiile de nivel mai înalt pe care le utilizați pentru a accesa fluxuri derivă din funcțiile membre din *streambuf*.

UN EXEMPLU SIMPLU DE STREAMBUF

C/C++ 1010

Așa cum ați învățat în secțiunea 1009, C++ prevede clasa de bază *streambuf* pentru a vă ajuta la manevrarea fluxurilor de intrare și ieșire. În timp ce programele dumneavoastră ar trebui să utilizeze în general clasele de nivel înalt și membrii lor pentru a controla fluxurile, puteți însă utiliza și o instanță a clasei *streambuf* în cadrul programelor dumneavoastră, tot așa cum ați fi utilizat o instanță a unei clase de nivel mai ridicat. Următorul program, *cu_sbuf.cpp* utilizează un obiect *streambuf* pentru a scrie textul introdus de la tastatură într-un fișier de pe disc:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    int c;
    const char *numefis = "_junk_.$$$";
    ofstream outfile;
    streambuf *out, *input = cin.rdbuf();
    // Se pozitioneaza la capatul fisierului. Adauga tot textul.
    outfile.open( numefis, ios::ate | ios::app);
    if (!outfile)
      {
        cerr << "Nu pot deschide " << numefis;
        return(-1);
      }
    out = outfile.rdbuf(); // Conecteaza ofstream si streambuf.
    clog << "Introduceti niste text. Utilizati Control-Z pentru a
              incheia." << endl;
    while ( (c = input -> sbumpc() ) != EOF)
      {
        cout << char(c); // Afiseaza pe ecran.
        if (out -> sputc(c) == EOF)
          cerr << "Eroare la iesire";
      }
  }
```

Atunci când compilați și executați programul *cu_sbuf.cpp*, el va deschide fișierul *_junk_.$$$* în directorul curent. Apoi programul atașează obiectul *out streambuf* bufferului pentru fișierul *junk*. Programul testează fiecare literă pe care o introduce utilizatorul la tastatură până la întâlnirea unei intrări CTRL+Z (EOF). Programul utilizează membrul *streambuf* pentru ieșirea fiecărei intrări diferite de EOF către ecran și către fișier. După cum puteți vedea, programul nu este mult diferit de programele pe care le-ați scris anterior, care utilizau funcțiile membre *get* și *put* și nu alte clase bazate pe fluxuri.

1011 CITIREA DATELOR BINARE UTILIZÂND FUNCȚIA READ

C/C++

Așa cum ați învățat în secțiunea 1009, programele dumneavoastră pot utiliza metoda *get* a clasei *istream* pentru a citi informații de câte un bit o dată dintr-un fișier binar. Programele dumneavoastră pot utiliza de asemenea și funcția membru *read* pentru a citi date binare dintr-un fișier, câte un bloc (un număr de octeți pe care l-ați dat dumneavoastră) o dată. În programele dumneavoastră veți utiliza funcția *read* ca mai jos:

```
#include <iostream.h>
#include <fstream.h>

istream &read(unsigned char *buffer, int num);
```

Funcția membru *read* citește *num* octeți din fluxul de intrare. Apoi funcția membru *read* plasează acești *num* octeți în bufferul de memorie care începe la adresa indicată de parametrul *buffer*. Parametrul *num* determină dimensiunea blocului pe care *read* îl returnează din fișier. În general, blocurile dumneavoastră ar trebui să fie relativ reduse ca dimensiune, pentru a preveni supraîncărcarea memoriei și pentru a păstra manevrabilitatea fișierului în eventualitatea că o operație de citire eșuează. Mai clar, metoda *read* este cu mult mai puternică și mai utilă decât *get* (deoarece puteți citi secvențe de date și nu numai simple caractere), punându-vă la dispoziție cunoștințe mai avansate despre construcția fișierului pe care programul dumneavoastră îl va citi.

1012 SCRIEREA DATELOR BINARE UTILIZÂND FUNCȚIA WRITE

C/C++

Așa cum ați învățat în secțiunea 1010, programele dumneavoastră pot utiliza funcția membru *put* a clasei *ostream* pentru a scrie date binare într-un fișier. Așa cum ați învățat în secțiunea 1011, programele dumneavoastră pot utiliza metoda *read* a clasei *istream* pentru a citi un număr predeterminat de octeți dintr-un fișier. În general, veți utiliza metoda *read* în combinație cu un fișier pe care l-ați scris utilizând funcția membru *write* a clasei *ostream*. Veți utiliza funcția membru *write* în programele dumneavoastră așa cum arătăm mai jos:

```
#include <iostream.h>
#include <fstream.h>

iostream &write(const unsigned char *buffer, int num);
```

Funcția *write* scrie numărul de octeți specificat de parametrul *num* – începând de la locația de memorie indicată de parametrul *buffer* – către fluxul cu care ați apelat funcția. Pentru a înțelege mai bine metodele *read* și *write*, analizați următorul program, *rw_struc.cpp* care scrie o structură pe disc și apoi citește din nou structura și o afișează:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

struct stare
  {
```

```
    char nume[80];
    float balanta;
    unsigned long nr_cont;
  };
void main(void)
  {
    struct stare cont;
    strcpy(cont.nume, "Lars Klander");
    cont.balanta = 1234.56;
    cont.nr_cont = 98765432;
    ofstream outbal("balanta.asc", ios::out | ios::binary);
    if(!outbal)
      {
        cout << "Nu pot deschide fisierul de iesire." << endl;
        exit (1);
      }
    outbal.write((unsigned char *) &cont, sizeof(struct stare));
    outbal.close();
    ifstream inbal("balanta.asc", ios::in | ios::binary);
    if(!inbal)
      {
        cout << "Nu pot deschide fisierul." << endl;
        exit (1);
      }
    inbal.read((unsigned char*) &cont, sizeof(struct stare));
    cout << cont.nume << endl;
    cout << "Numarul contului: " << cont.nr_cont <<endl;
    cout.precision(2);
    cout.setf(ios::fixed);
    cout << "Balanta: $" << cont.balanta << endl;
    inbal.close();
  }
```

Programul scrie o singură înregistrare de tipul *stare* în fișierul *balanta.asc*. După cum puteți vedea, fișierul *balanta.asc* conține numele unui cont, balanța de plăți curentă a contului, precum și numărul de cont. Programul scrie valorile conținute de obiectul de tip *stare* în fișierul *balanta.asc* ca pe o serie de valori binare. Apoi programul închide fluxul de ieșire și deschide un flux de intrare. Fluxul de intrare citește valorile binare salvate într-o altă instanță, diferită, a structurii *stare*. În final, programul afișează conținutul fișierului înapoi pe ecran pentru ca utilizatorul să-l poată verifica.

UTILIZAREA FUNCȚIEI MEMBRU GCOUNT

C/C++1013

După cum ați învățat în secțiunea 1011, programele dumneavoastră pot utiliza metoda *read* din clasa *istream* pentru a citi un anumit număr de octeți dintr-un fișier. Uneori totuși o operație *read* poate eșua și dumneavoastră trebuie să determinați câte caractere a obținut din flux funcția *read*, înainte de a se opri. Pentru aceasta, programele dumneavoastră pot

utiliza funcția membru *gcount*, pe care o veți utiliza în cadrul programelor dumneavoastră cum arătăm în continuare:

```
#include <iostream.h>
#include <fstream.h>
int gcount(void);
```

Atunci când apelați funcția membru *gcount* cu un flux de intrare, funcția *gcount* va returna numărul de caractere citite de ultima operație binară de intrare. Pentru a înțelege mai bine acest proces, analizați următorul program, *gcount.cpp*, care ilustrează cum puteți utiliza funcția *gcount*:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
  {
    float fnum[4] = {99.75, -34.4, 1776.0, 200.1};
    int i;
    ofstream out("numere.asc", ios::out | ios::binary);
    if(!out)
      {
        cout << "Nu pot deschide fisierul.";
        exit (1);
      }
    out.write((unsigned char *) &fnum, sizeof(fnum));
    out.close();
    for (i=0; i<4; i++)
      fnum[i] = 0.0;
    ifstream in("numere.asc", ios::in | ios::binary);
    if(!in)
      {
        cout << " Nu pot deschide fisierul.";
        exit (1);
      }
    in.read((unsigned char *) &fnum, sizeof(fnum));
    cout << in.gcount() << " octeti cititi." << endl;
    for (i=0; i<4; i++)
      cout << fnum[i] << " ";
    in.close();
  }
```

Programul *gcount.cpp* definește o matrice de patru valori în virgulă mobilă. Apoi programul scrie cele patru valori într-un fișier ASCII. După aceea programul închide fluxul de ieșire pe care l-a folosit pentru a scrie fișierul ASCII. După ce închide fluxul, programul șterge valorile din matricea *fnum* și deschide un flux de intrare pentru a citi din nou valorile din fișier. Programul vă atenționează asupra numărului de octeți pe care i-a citit, apoi afișează valorile în virgulă mobilă citite de pe disc. Atunci când compilați și executați programul *gcount.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
16 octeti cititi
99.75 -34.4 1776 200.1
C:\>
```

UTILIZAREA FUNCȚIILOR GET SUPRAÎNCĂRCATE

C/C++ 1014

După cum ați învățat, programele dumneavoastră pot utiliza funcția *get*, membru al clasei *istream*, pentru a citi câte un singur caracter o dată dintr-un fișier text. Dar majoritatea compilatoarelor de C++ conțin de asemenea două versiuni supraîncărcate ale metodei *get*. Prima versiune supraîncărcată citește o serie de caractere; cea de-a doua citește câte o singură valoare *int* o dată. Veți utiliza versiunile supraîncărcate ale metodei *get* în cadrul programelor dumneavoastră după cum urmează:

```
#include <iostream.h>
#include <fstream.h>

ostream &get(char *buffer, int num, char delimitator ='\n');
int get(void);
```

Prima funcție supraîncărcată citește caractere în șirul de caractere către care indică parametrul buffer. Funcția citește caractere până când se întâmplă unul din evenimentele de mai jos: fie funcția citește numărul de caractere specificat de parametrul *num*, fie întâlnește caracterul specificat de parametrul *delimitator*. (Parametrul *delimitator* este implicit caracterul de linie nouă.) Dacă funcția supraîncărcată *get* întâlnește delimitatorul, ea va opri citirea imediat și fără a îndepărta delimitatorul din fluxul de intrare.

Cea de-a doua funcție supraîncărcată returnează următorul caracter din flux ca pe o valoare întreagă. Ea va returna constanta EOF dacă întâlnește marcajul de sfârșit de fișier. Cea de-a doua funcție supraîncărcată ar trebui să vă pară familiară, fiind similară funcției C *getc*.

UTILIZAREA METODEI GETLINE

C/C++ 1015

În secțiunile precedente ați învățat câteva metode diferite pe care programele dumneavoastră le pot utiliza pentru a citi informații dintr-un flux de intrare. De asemenea, limbajul C++ pune la dispoziție și metoda *getline*, care permite programelor dumneavoastră să citească date dintr-un fișier câte o linie o dată. Veți utiliza această metodă în programele dumneavoastră după cum arătăm mai jos:

```
#include <iostream.h>
#include <fstream.h>
ostream &getline(char *buffer, int num, char delimitator ='\n');
```

După cum puteți vedea, metoda *getline* este perfect identică primei metode *get* supraîncărcate. Metoda *getline* citește date până când întâlnește delimitatorul *delimitator* sau până când citește numărul de caractere specificat de parametrul *num*. Metoda *getline* plasează caracterele citite în bufferul care începe la pointerul *buffer*. De exemplu, următorul program, *getline.cpp*, citește conținutul unui fișier text câte o linie o dată și o afișează pe ecran:

```
#include <iostream.h>
#include <fstream.h>

void main(int argc, char *argv[])
  {
    if(argc!=2)
      {
        cout << "Apelare: PR <numefisier>" << endl;
        exit (1);
      }
    ifstream in(argv[1]);
    if(!in)
      {
        cout << "Nu pot deschide fisierul.";
        exit (1);
      }
    char sir[255];
    while(in)
      {
        in.getline(sir, 255); // delimitatorul implicit - linia noua
        cout << sir << endl;
      }
    in.close();
  }
```

1016 DETECTAREA SFÂRȘITULUI DE FIȘIER

C/C++

După cum ați învățat, atunci când lucrați cu fluxuri, programele dumneavoastră pot determina când indicatorul de fișier atinge marcatorul de sfârșit de fișier prin compararea cu constanta EOF a valorii pe care instrucțiunea *get* sau o funcție alternativă o returnează. Programele dumneavoastră pot de asemenea să testeze dacă a fost întâlnit marcatorul de sfârșit de fișier prin testarea valorii metodei *eof*, pe care o veți utiliza în programele dumneavoastră ca mai jos:

```
#include <iostream.h>
#include <fstream.h>

int eof(void);
```

Funcția *eof* returnează 0 (fals) în mod normal și o valoare diferită de zero dacă indicatorul de fișier se află la sfârșitul fișierului. Următorul program, *test_eof.cpp* folosește funcția *eof* pentru a citi un fișier dintr-un flux:

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>
```

```
void main(int argc, char *argv[])
  {
    if(argc!=2)
      {
        cout << "Apel: test_eof <numefisier>" << endl;
        exit (1);
      }
    ifstream in(argv[1], ios::in | ios::binary);
    if(!in)
      {
        cout << "Nu pot deschide fisierul de intrare." << endl;
        exit (1);
      }
    register int i, j;
    int nr = 0;
    char c[16];
    cout.setf(ios::uppercase);
    while(!in.eof())
      {
        for(i=0; i<16 && !in.eof(); i++)
          in.get(c[i]);
        if(i<16)
          i--; // Nu tipareste EOF
        for(j=0; j < i; j++)
          cout << setw(3) << hex << (int) c[j];
        for(; j < 16; j++)
          cout << "   ";
        cout << "\t";
        for(j=0; j < i; j++)
          if(isprint(c[j]))
            cout << c[j];
          else
            cout << ".";
        cout << endl;
        nr ++;
        if(nr==16)
          {
            nr = 0;
            cout << "Apasati ENTER pentru a continua: ";
            cin.get();
            cout << endl;
          }
      }
    in.close();
  }
```

Programul *test_eof.cpp* citește dintr-un fișier de intrare câte 16 octeți o dată. De fiecare dată când programul citește 16 octeți, el afișează acești 16 octeți pe ecran sub forma de secvență

de numere hexazecimale. Apoi, programul afişează aceiaşi 16 octeţi pe ecran ca secvenţă de litere. După ce programul efectuează acest proces de afişare de 16 ori, el se opreşte şi aşteaptă ca utilizatorul să apese o tastă. Apoi citeşte 16 octeţi de încă 16 ori (în total vor fi 256 de octeţi la fiecare ciclu) până când va citi marcatorul de sfârşit de fişier. Atunci când compilaţi şi executaţi programul *test_eof.cpp* împreună cu fişierul *test_eof.cpp*, programul va afişa următorul output, parţial afişat în continuare:

```
23 69 6E 63 6C 75 64 65 20 3C 69 6F 73 74 72 65 #include <iostr
61 6D 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20 eam.h>..#include
3C 66 73 74 72 65 61 6D 2E 68 3E  D  A 23 69 6E <fstream.h>..#in
63 6C 75 64 65 20 3C 63 74 79 70 65 2E 68 3E  D clude <ctype.h>.
 A 23 69 6E 63 6C 75 64 65 20 3C 69 6F 6D 61 6E .#include <ioman
69 70 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20 ip.h>..#include
3C 73 74 64 69 6F 2E 68 3E  D  A  D  A 69 6E 74 <stdio.h>.
20 6D 61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 ..int main(int a
63 68 61 72 20 2A 61 72 67 76 5B 5D 29  D  A 20 rgc, char *argv[
7B  D  A 20 20 20 69 66 28 61 72 67 63 21 3D 20 ]).. {.. if(ar
29  D  A 20 20 20 20 7B  D  A 20 20 20 20 20 20 gc!=2).. {..
63 6F 75 74 20 3C 3C 20 55 73 61 67 65 3A 20 3A cout << "Ap
65 6C 3A 20 74 65 73 74 5F 65 6F 66 20 3C 6E 75 el: test_eof <nu
6D 65 66 69 73 69 65 72 3E 22 20 3C 3C 20 65 6E mefisier>" << en
64 6C 3B  D  A 20 20 20 20 20 20 65 78 69 74 20 dl;.. exit
Apasati ENTER pentru a continua:
```

1017 UTILIZAREA FUNCŢIEI IGNORE

C/C++

După cum aţi învăţat C++ vă pune la dispoziţie multe instrumente pe care le puteţi utiliza pentru a citi sau scrie din şi în fluxurile de intrare-ieşire. Totuşi, nici-una dintre funcţiile despre care aţi învăţat până acum nu vă permit să citiţi şi apoi să eliminaţi caractere din fluxul de intrare. Pentru a efectua operaţii de citire şi eliminare, programele dumneavoastră pot utiliza funcţia *ignore*, pe care o veţi utiliza aşa cum arătăm mai jos:

```
#include <iostream.h>
#include <fstream.h>

istream &ignore(int num=1, int delimitator =EOF);
```

Funcţia *ignore* citeşte şi elimină caractere până când fie a îndepărtat *num* caractere din fluxul de intrare, fie a întâlnit parametrul *delimitator*, care este *EOF* în mod implicit. Dacă funcţia ignore a întâlnit delimitatorul, ea nu va îndepărta caracterul *delimitator* din fluxul de intrare. În schimb, funcţia *ignore* îşi va opri procesarea imediat. Următorul program, *ignorec.cpp* se citeşte pe sine însuşi de pe disc. El ignoră caracterele până când întâlneşte un spaţiu sau până când citeşte 10 caractere şi apoi programul afişează restul fişierului:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    ifstream in("ignorec.cpp");
    if(!in)
      {
```

```
        cout << "Nu pot deschide fisierul." << endl;
        exit (1);
      }
    in.ignore(10, ' ');
    char c;
    while(in)
      {
        in.get(c);
        cout << c;
      }
    in.close();
  }
```

Codul programului *ignorec.cpp* utilizează funcția membru *ignore* pentru a sări peste primele 10 caractere sau până la primul spațiu, în funcție de ce apare mai întâi, din cadrul fișierului denumit. Apoi programul utilizează funcția *get* pentru a citi restul caracterelor rămase în fișier. Programul afișează pe ecran fiecare caracter pe care îl citește.

UTILIZAREA FUNCȚIEI PEEK

C/C++1018

După cum ați învățat, C++ vă furnizează multe funcții pentru accesarea fluxurilor de intrare. Atunci când programele dumneavoastră lucrează cu fluxuri de intrare, probabil că uneori veți avea nevoie să „vedeți" următorul caracter dintr-un flux, fără a mai îndepărta acel caracter din flux. Pentru a face aceasta, programele dumneavoastră pot utiliza metoda *peek*, pe care o veți utiliza în cadrul programelor dumneavoastră ca mai jos:

```
#include <iostream.h>
#include <fstream.h>

int peek(void);
```

Funcția *peek* returnează următorul caracter din flux sau constanta EOF dacă pointerul de fișier este la capătul fișierului.

UTILIZAREA FUNCȚIEI PUTBACK

C/C++1019

Atunci când lucrați cu fluxuri, uneori va fi nevoie ca programele dumneavoastră să returneze un caracter pe care l-au citit dintr-un flux de intrare. Puteți utiliza funcția *putback* în programele dumneavoastră pentru a returna ultimul caracter pe care programul l-a citit dintr-un flux de intrare dat către acel flux. În programele dumneavoastră veți utiliza funcția *putback* ca mai jos:

```
#include <iostream.h>
#include <fstream.h>

istream &putback(char c);
```

Funcția *putback* plasează caracterul *c* înapoi în fluxul pe care l-a apelat programul și returnează o referință către acel flux. Observați că *putback* returnează doar un caracter către fluxul din care programul l-a preluat; nu veți putea folosi funcția *putback* pentru a plasa caractere în alte fluxuri decât cele din care programul a citit inițial caracterele.

1020 DETERMINAREA POZIȚIEI CURENTE DINTR-UN FLUX

C/C++

După cum ați învățat, în multe cazuri programele dumneavoastră vor să cunoască poziția curentă a indicatorului de fișier. În secțiunea 1016 ați utilizat funcția membru *eof* pentru a determina dacă pointerul de fișier se afla la capătul fișierului. Însă programele dumneavoastră trebuie să știe frecvent unde se află în cadrul unui fișier. Atunci când programele dumneavoastră trebuie să determine poziția curentă a indicatorului de fișier, ele pot utiliza membrii *tellg* și *tellp*, sub forma arătată mai jos:

```
#include <iostream.h>
#include <fstream.h>

long output.tellp(void);
long input.tellg(void);
```

Este important să observați că *tellp* și *tellg* efectuează același lucru. Totuși programele dumneavoastră vor trebui să utilizeze întotdeauna *tellp* cu fluxuri de ieșire și *tellg* cu fluxuri de intrare.

1021 CONTROLUL INDICATORULUI DE FIȘIER FLUX

C/C++

După cum ați învățat, fișierele flux *ifstream* și *ofstream* au funcții membre pe care programele dumneavoastră le pot utiliza pentru a efectua operații de intrare-ieșire. Atunci când programele dumneavoastră efectuează astfel de operații, uneori va fi nevoie ca ele să stabilească poziția indicatorului de fișier flux. Pentru a poziționa indicatorul de fișier, programele dumneavoastră pot utiliza metodele *seekg* (pentru intrări) și *seekp* (pentru ieșiri), așa cum arătăm mai jos:

```
#include <iostream.h>
#include <fstream.h>

istream &seekg(streamoff deplasament [, seek_dir origine]);
ostream &seekp(streamoff deplasament [, seek_dir origine]);
```

Deplasamentul octeților este o valoare de tipul *long* (specificată în cadrul fișierului antet *iostream.h*) pe care, dacă nu specificați altfel cu ajutorul parametrului opțional origine, C++ o va aplica pornind cu începutul fișierului. Pentru a aplica deplasamentul biților începând de la altă locație decât începutul fișierului, puteți utiliza una din următoarele valori pentru parametrul origine:

```
enum seek_dir { beg=0, cur=1, end=2 };
```

1022 UTILIZAREA FUNCȚIILOR SEEKG ȘI SEEKP PENTRU ACCES ALEATOR

C/C++

În secțiunea 1021 ați învățat despre metodele *seekg* și *seekp* pe care programele dumneavoastră le pot utiliza pentru a manevra indicatorii de fișier în interiorul unui fișier cu acces aleator. Pentru a înțelege mai bine cum se utilizează metodele *seekg* și *seekp* într-un program, analizați următorul program, *schimba.cpp* care utilizează funcția *seekp* pentru a scrie peste un anumit caracter dintr-un fișier:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
   if(argc!=4)
     {
       cout << "Apel: schimba <numefisier> <octet> <caracter>"
          << endl;
       exit (1);
     }
   fstream out(argv[1], ios::in | ios::out | ios::binary);
   if(!out)
     {
       cout << "Nu pot deschide fisierul!";
       exit (1);
     }
   out.seekp(atoi(argv[2]), ios::beg);
   out.put(*argv[3]);
   out.close();
}
```

Atunci când compilaţi şi executaţi programul *schimba.cpp*, el caută locaţia specificată de dumneavoastră în parametrul *octet* din linia de comandă. Apoi el înlocuieşte acest octet cu caracterul specificat de dumneavoastră în parametrul liniei de comandă denumit *caracter*.

Observaţie: *Dacă specificaţi o valoare pentru* ***octet*** *care este dincolo de indicatorul* ***EOF***, *programul va scrie un caracter la acel octet şi va muta indicatorul* ***EOF*** *la acea locaţie. Dacă intenţionaţi să utilizaţi un program care utilizează procesări similare cu* ***schimba.cpp*** *pentru scopuri ceva mai avansate, faceţi un test în program care să verifice că valoarea* ***octet*** *nu este mai mare decât lungimea fişierului.*

MANEVRAREA POZIŢIEI INDICATORULUI DE FIŞIER

C/C++1023

După cum aţi învăţat, clasele din *iostream.h* vă oferă un control semnificativ şi informaţii în legătură cu poziţia indicatorului de fişier şi activităţile pe care le desfăşoară în cadrul fişierului. Atunci când combinaţi informaţiile de acces aleator cu cele pe care fluxurile le pot returna programelor dumneavoastră, aveţi un instrument puternic de manevrare a fişierelor cu acces aleator. Următorul program, *seek_test.cpp* se mută în diferite locaţii dintr-un fişier, spunându-vă poziţia sa curentă din fişier şi afişează fişierul astfel manevrat atunci când şi-a încheiat activitatea. Deşi programul *seek_test.cpp* nu îndeplineşte multe funcţii, el face mai clară relaţia dintre indicatorul de fişier şi datele din fişier, ca mai jos:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
```

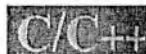
```
void main(int argc, char *argv[])
  {
    if(argc!=4)
      {
        cout << "Apel: seek_test <numefis> <octet> <caracter>"
          << endl;
        exit(1);
      }
    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out)
      {
        cout << "Nu pot deschide fisierul!";
        exit(1);
      }
    out.seekp(atoi(argv[2]), ios::beg);
    out.put(*argv[3]);
    cout << "Pozitia curenta e: " << out.tellp() <<endl;
    out.close();
  }
```

Programul *seek_test.cpp* primește 4 parametri de la utilizator în linia de comandă: numele programului, fișierul pe care operează, numărul de octeți cu care va deplasa indicatorul pornind de la începutul fișierului și un caracter care să fie plasat la acea locație în fișier. Apoi programul se mută la respectiva locație din fișier, pune la ieșire caracterul, vă informează despre ceea ce a făcut și care este poziția sa curentă. Veți rula programul *seek_test.cpp* cu instrucțiunile liniei de comandă similare cu următoarea:

```
C:\>seek_test text.txt 10 A
```

Observație: *Ca și în secțiunea 1022, dacă specificați o valoare a parametrului* ***octet*** *care se află dincolo de indicatorul* ***EOF****, programul va scrie un caracter în acel octet și va muta indicatorul de* ***EOF*** *la acea locație. Dacă intenționați să utilizați un program care utilizează procesări similare cu* ***seek_test.cpp*** *pentru operații ceva mai avansate, faceți un test în program care să verifice că valoarea octet nu este mai mare decât lungimea fișierului.*

1024 DETERMINAREA STĂRII CURENTE A UNUI FLUX DE INTRARE/IEȘIRE



Sistemul de intrări și ieșiri din C++ pune la dispoziție informații despre starea fiecărei operații de intrare/ieșire pe care o efectuează programele dumneavoastră.

C++ păstrează starea curentă a sistemului de intrări și ieșiri într-un întreg a cărui valoare programele dumneavoastră o pot accesa prin apelarea funcției membre *rdstate*. Acest întreg cuprinde indicatoarele pe biți arătate în tabelul 1024.

Indicator	Semnificație
eofbit	1 atunci când indicatorul de fișier atinge marcajul de sfârșit de fișier 0 altfel
failbit	1 atunci când apare o eroare posibil nefatală 0 altfel
badbit	1 atunci când apare o eroare de I/O fatală 0 altfel

***Tabelul 1024** Indicatoarele pe biți specificate de către ios*

În cadrul programelor dumneavoastră, puteți utiliza funcția membru *rdstate* pentru a testa starea curentă a intrărilor și ieșirilor. Veți implementa funcția membru *rdstate* ca mai jos:

```
#include <iostream.h>
#include <fstream.h>

int rdstate(void)
```

Pentru a testa starea curentă a unui flux de intrare/ieșire, veți utiliza un cod similar celui din fragmentul următor:

```
stare = in.rdstate();
if(stare & ios::eofbit)
   cout <<< " A intilnit EOF." << endl;
if(stare & ios::failbit)
   cout << " Eroare nefatala." << endl;
if(stare & ios::badbit)
   cout << " Eroare fatala." << endl;
```

Totuși, ca și în cazul tratărilor de excepții, programele dumneavoastră probabil că vor efectua procesări mult mai utile decât simpla afișare în cazul unei erori de acest gen. Nu trebuie să construiți nici o buclă *while* care nu permite utilizatorului să iasă până când nu sunt rezolvate erorile sau este selectată operațiunea de anulare dintr-o listă de opțiuni. Ca și în cazul tratărilor de erori, care permit programelor dumneavoastră să efectueze procesări mai complexe cu un risc mai mic de a se produce o eroare fatală de sistem, tot așa, testarea constantă a valorii returnate de funcția *rdstate* vă ajută să vă protejați programul împotriva erorilor de I/O care pot duce până la căderea sistemului.

CLASELE DE MATRICE DE INTRARE-IEȘIRE

C/C++1025

C++ acceptă trei clase de I/O bazate pe matrice, corespunzătoare claselor I/O bazate pe fișiere. Cele trei clase bazate pe matrice sunt : *istrstream, ostrstream* și *strstream*. C++ derivă toate aceste trei clase din *strstreambuf* printre alte clase de bază. Clasa de bază *strstreambuf* definește câteva detalii de nivel jos pe care clasele derivate le utilizează. În plus față de *strstreambuf*, C++ mai derivă clasa *istrstream* și din *istream*, *ostream* și *strstream*.

Din cauza locului lor în cadrul structurii de moștenire, toate cele trei clase de fluxuri bazate pe matrice au acces la aceleași funcții membre ca și clasele de fluxuri bazate pe fișiere. Puteți prin urmare să utilizați în programele dumneavoastră matrice de fluxuri pentru a efectua o mare parte a operațiunilor pe care le-ați fi efectuat într-un flux fișier.

Observație: *Clasele de I/O bazate pe matrice permit programelor dumneavoastră în C++ să efectueze activități similare celor pentru care programele dumneavoastră în C utilizau sprintf. Clasele de I/O bazate pe matrice permit programelor dumneavoastră să stocheze în buffer și să formateze datele de ieșire înainte de a fi afișate.*

1026 FLUXURILE DE TIP ȘIR DE CARACTERE

C/C++

În capitolele dedicate limbajului C ale acestei cărți, ați învățat că programele dumneavoastră pot utiliza funcțiile *sprintf* și *sscanf* pentru a scrie și a citi date în și dintr-un șir de caractere. Pentru a vă ajuta să efectuați operații similare, fișierul antet *ostrstream.h* definește clasa *ostrstream*. Atunci când programele dumneavoastră creează un flux de ieșire de tip șir de caractere, de fapt veți îndrepta acel flux către un anumit șir de caractere. Următorul program, *umplestr.cpp* creează o variabilă de tipul *ostrstream* și o va completa cu caracterele „Jamsa's C/C++ Programmer's Bible":

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char sir[256];
    ostrstream sr(sir, 256);  // Limiteaza sirul de caractere
    sr << "Jamsa's C/C++ Programmer's Bible" << ends;
    cout << sir;
  }
```

1027 UTILIZAREA CLASEI ISTRSTREAM PENTRU SCRIEREA UNUI ȘIR DE CARACTERE

C/C++

În secțiunea 1026 ați învățat cum să declarați o variabilă de tipul *ostrstream* și cum să utilizăm acest flux șir pentru a afișa informații din interiorul programelor dumneavoastră. Bibliotecile C++ definesc de asemenea și clasa *istrstream*, pe care programele dumneavoastră o pot utiliza pentru a îndrepta un flux de intrare către o matrice. În cadrul programelor dumneavoastră, veți declara tipul *istrstream* ca mai jos:

```
#include <iostream.h>
#include <strstrea.h>

istrstream istr(char *buffer);
```

În constructorul clasei *istrstream*, parametrul *buffer* este un pointer către o matrice pe care fluxul o va folosi ca sursă pentru caractere. Pentru a vă ajuta să înțelegeți mai bine procesările pe care le efectuează clasa *istrstream*, analizați următorul program, *primul_in.cpp*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char in_sir[] = "10 Salut 0x88 12.23 gata";
```

```
    istrstream ins(in_sir);
    int i;
    char sr[80];
    float f;
    ins >> i;
    ins >> sr;
    cout << i << " " << sr << endl;
    ins >> i;
    ins >> f;
    ins >> sr;
    cout << hex << i << " " << f << " " << sr;
}
```

Deoarece fluxurile de intrare din C++ nu mai acceptă date după ce au întâlnit spaţii albe, puteţi utiliza operatorul de inserare cu şirul de caractere *in_sir* pentru a completa celelalte variabile pe care programul le defineşte mai departe pe parcursul execuţiei. Atunci când se execută programul, el va completa variabila *i* cu valoarea 10 şi variabila *sr* cu valoarea "Salut." Programul afişează apoi aceste valori pe prima linie. După ce programul generează prima linie a ieşirii, el completează variabila *i*, de această dată cu valoarea 0x88. Apoi, programul completează variabila *f* cu valoarea 12.23 şi *sr* cu "gata.". Atunci când compilaţi şi executaţi programul *primul_in.cpp*, pe ecranul dumneavoastră se vor afişa următoarele:

```
10 Salut
88 12.23 gata
C:\>
```

CLASA OSTRSTREAM

C/C++1028

După cum aţi învăţat în secţiunea 1026, programele dumneavoastră pot utiliza matricele de ieşire ale clasei *ostrstream* pentru a formata şi a proiecta ieşirile înainte de a afişa ieşirea. Atunci când declaraţi o instanţă a clasei *ostrstream* în cadrul programelor dumneavoastră, veţi utiliza următoarea formă generală:

```
#include <iostream.h>
#include <strstrea.h>

ostrstream ostr(char *buffer, int dimensiune, int mod=ios::out);
```

Parametrul *buffer* conţine adresa de start a matricei către care *ostr* scrie şirul de ieşire. Parametrul *dimensiune* conţine dimensiunea în octeţi a bufferului. În sfârşit, parametrul opţional *mod* vă permite să controlaţi modul în care C++ deschide fluxul. În mod implicit, C++ deschide fluxul pentru ieşire normală, dar dumneavoastră puteţi, de exemplu, să modificaţi fluxul astfel încât C++ să adauge în mod automat toate ieşirile la flux.

Atunci când declaraţi un flux ca fiind bazat pe matrice, programele dumneavoastră vor scrie toate informaţiile direcţionate către flux în matrice. Însă, trebuie să aveţi grijă ca fluxurile dumneavoastră să nu depăşească dimensiunea bufferului, altfel programul va returna o eroare sau chiar va provoca o cădere a sistemului.

1029 UTILIZAREA FORMELOR SUPRAÎNCĂRCATE PENTRU ISTRSTREAM

C/C++

După cum ați învățat în secțiunea 1027, de obicei veți declara un flux de intrare bazat pe matrice cu numai un parametru, un pointer către șirul care deține caracterele de intrare ale fluxului. Totuși, C++ acceptă și o versiune supraîncărcată a constructorului clasei *istrstream*, a cărui formă generală este arătată în continuare:

```
#include <iostream.h>
#include <strstrea.h>

istrstream istr(char *buffer, int dimensiune);
```

Puteți utiliza constructorul supraîncărcat pentru a limita accesul fluxului doar la primele *dimensiune* elemente din matricea către care indică parametrul *buffer*. Veți utiliza constructorul supraîncărcat atunci când dețineți în avans informații despre matrice sau atunci când doar aveți nevoie de primele *dimesiune* elemente și ignorați restul. De exemplu, următorul program, *stre_in.cpp* utilizează versiunea supraîncărcată a constructorului clasei *istrstream*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char in_sir[] = "10 Salut 0x88 12.23 gata";
    istrstream ins(in_sir, 8);
    int i;
    char sr[80];
    float f;
    ins >> i;
    ins >> sr;
    cout << i << " " << sr << endl;
    ins >> i;
    ins >> f;
    ins >> sr;
    cout << hex << i << " " << f << " " << sr;
  }
```

Programul *stre_in.cpp* limitează dimensiunea șirului pe care fluxul îl poate accesa la primii 8 biți. Prima citire execută exact aceleași lucruri ca în programul *primul_in.cpp*, prezentat în secțiunea 1027, afișând 10 și "Salut". Totuși, din cauză că fluxul nu mai poate accesa șirul (deoarece a atins limita desemnată a matricei) după primii 8 octeți, fluxul începe să citească date reziduale din puncte necunoscute din memorie. Atunci când compilați și executați programul *stre_in.cpp*, ecranul dumneavoastră va afișa următoarele:

```
10 Hello
a 5.9801e-39
C:\>
```

UTILIZAREA FUNCȚIEI PCOUNT CU MATRICE DE IEȘIRE

C/C++1030

Atunci când programele dumneavoastră lucrează cu matrice de ieșire, uneori programele dumneavoastră pot avea nevoie să cunoască câte caractere se găsesc în matricea de ieșire. C++ vă permite să utilizați funcția membru *pcount* pentru a determina numărul de caractere pe care îl conține o matrice de ieșire. Veți implementa funcția membru *pcount* ca mai jos:

```
#include <iostream.h>
#include <strstrea.h>

int pcount(void);
```

Dacă matricea de ieșire conține terminatorul *NULL*, funcția *pcount* va include terminatorul în cadrul valorii pe care o returnează (cu alte cuvinte, dacă șirul este lung de 15 caractere plus terminatorul *NULL*, *pcount* va returna valoarea 16). Puteți utiliza *pcount* împreună cu matrice de ieșire pentru a controla mai bine procesele programului. Programul *pcount.cpp* utilizează funcția *pcount* pentru a afișa numărul de caractere dintr-un flux de ieșire, ca mai jos:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char sr[80];
    ostrstream outs(sr, sizeof(sr));
    outs << "Salut ";
    outs << 34 << " " << 9876.98;
    outs << ends;
    cout << "Lungimea sirului: " << outs.pcount() << endl;
    cout << sr;
  }
```

Programul *pcount.cpp* adaugă câteva caractere și valori la fluxul *outs*. După ce adaugă aceste informații la flux, el afișează lungimea fluxului și apoi fluxul însuși. Atunci când compilați și executați programul *pcount.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Lungimea sirului:
18
Salut 34   9876.98
C:\>
```

MANEVRAREA FLUXURILOR MATRICE CU FUNCȚIILE MEMBRE DIN IOS

C/C++1031

Atunci când programele dumneavoastră lucrează cu fluxuri bazate pe matrice, veți vedea că puteți utiliza funcțiile membre standardizate ale clasei *ios*, cum ar fi *get*, *put*, *rdstate*, *eof* și așa mai departe. De exemplu, următorul program, *tabl_get.cpp* citește conținutul unui flux matrice utilizând metoda *get*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char sr[] ="abcdefghijklmnop";
    istrstream ins(sr);
    char ch;
    while(!ins.eof())
      {
        ins.get(ch);
        cout << ch << " ";
      }
  }
```

Programul *tabl_get.cpp* atașează fluxul *ins* la matricea *sr*, apoi utilizează o buclă *while* pentru a citi din matricea *sr* câte un caracter o dată și pentru a afișa caracterul pe ecran. Atunci când compilați și executați programul *tabl_get.cpp*, ecranul dumneavoastră va afișa următoarele:

```
a b c d e f g h i j k l m n o p
C:\>
```

1032 UTILIZAREA CLASEI STRSTREAM

C/C++

În secțiunile precedente ați învățat cum să creați fluxuri bazate pe matrice atât pentru intrare, cât și pentru ieșire. Totuși, uneori programele dumneavoastră trebuie să creeze un singur flux pentru a manevra atât intrările cât și ieșirile. Programele dumneavoastră pot declara un singur flux care să manevreze toate operațiile de I/O utilizând constructorul *strstream*. În cadrul programelor dumneavoastră, veți declara obiectele de tipul *strstream* ca mai jos:

```
#include <iostream.h>
#include <strstrea.h>

strstream iostr(char *buffer, int dim, int mod);
```

Ca și în cazul fluxurilor de intrare și de ieșire, *buffer* este un pointer către începutul șirului, pe când *dim* reprezintă numărul de octeți din *buffer*. Puteți stabili modul utilizând valorile standard specificate în clasa *ios*. Însă, în general, veți fixa modul la *ios::in\ios:out*. În plus, trebuie să încheiați cu *NULL* matricele dumneavoastră. Pentru a înțelege mai bine modul de implementare al fluxurilor de I/O bazate pe matrice, analizați următorul program, *matrice_io.cpp* care folosește un singur flux pentru a efectua intrări și ieșiri către o matrice:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char iostr[80];
    strstream ios(iostr, sizeof(iostr), ios::in | ios::out);
    char sr[80];
    int a, b;
```

```
    ios << "10 20 testaretestare";
    ios >> a >> b >> sr;
    cout << a << " " << b << " " << sr << endl;
  }
```

Programul *matrice_io.cpp* declară o matrice de lungime 80, apoi atașează noul flux *ios* la această matrice. Cu ajutorul variabilei *ios*, programul citește mai întâi valori în flux, apoi depune valorile din matrice în variabile și afișează variabilele pe ecran. Atunci când compilați și executați programul *matrice_io.cpp*, ecranul dumneavoastră va afișa următoarele:

```
10 20 testaretestare
C:\>
```

EFECTUAREA ACCESULUI ALEATOR CU O MATRICE FLUX

C/C++ 1033

După cum ați învățat, puteți utiliza toate operațiile standard de I/O (cum ar fi formatarea și generarea de șiruri de caractere) împreună cu o matrice flux. Prin urmare, puteți utiliza metodele *seekg* și *seekp* pentru a manevra fluxuri matrice, ca și cu fluxurile bazate pe fișiere. Următorul program, *tabl_seek.cpp* utilizează funcția *seekg* pentru a modifica poziția indicatorului de flux în interiorul unei matrice flux:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char nume[]="Jamsa's C/C++ Programmer's Bible";
    char iostr[80];
    strstream ios(iostr, sizeof(iostr), ios::in | ios::out);
    char ch;
    ios << nume;
    ios.seekg(7, ios::beg);
    ios >> ch;
    cout << "Nume: " << nume << endl;
    cout << "Caracterul de pe pozitia 8: " << ch;
  }
```

Atunci când compilați și executați programul *tabl_seek.cpp*, ecranul dumneavoastră va afiș următoarele:

```
Nume: Jamsa's C/C++ Programmer's Bible
Caracterul de pe pozitia 8: C
C:\>
```

UTILIZAREA MANIPULATORILOR CU FLUXURILE MATRICE

C/C++ 1034

După cum ați învățat, C++ vă permite să tratați fluxul matrice ca și cum ar fi un flux obișnu bazat pe fișiere. Veți utiliza manipulatorii împreună cu fluxurile matrice în aceeași manieră î care le-ați utilizat anterior cu fluxuri fișier. Următorul program, *tabl_sag.cpp* folosește

pereche de manipulatori personalizați (*st* pentru săgeata stânga și *dr* pentru săgeata dreapta) asupra unui flux matrice:

```
#include <iostream.h>
#include <strstrea.h>

ostream &ra(ostream &flux)
  {
    flux << "-------->";
    return flux;
  }
ostream &la(ostream &flux)
  {
    flux << "<--------";
    return flux;
  }
void main(void)
  {
    char sr[80];
    ostrstream outs(sr, sizeof(sr));
    outs << ra << " Uitati-va la acest numar: ";
    outs << 100000 << la << ends;
    cout << " " << sr << endl;
  }
```

Atunci când compilați și executați programul *tabl_sag.cpp*, ecranul dumneavoastră va afișa următoarele:

```
-------->Uitati-va la acest numar: 100000<------
C:\>
```

1035 UTILIZAREA UNUI OPERATOR DE INSERARE PERSONALIZAT CU FLUXURILE MATRICE

C/C++

După cum ați învățat, C++ vă permite să tratați fluxurile bazate pe matrice ca și pe cele bazate pe fișiere. Deoarece fluxurile matrice sunt identice în mare măsură cu cele bazate pe fișiere, dumneavoastră puteți crea proprii operatori de inserare și de extragere pentru fluxurile matrice, de aceeași manieră în care ați făcut-o pentru fluxurile fișier. De exemplu, următorul program, *pct_tabl.cpp* creează o clasă denumită *pct* și utilizează operatori personalizați de inserare pentru a afișa datele clasei:

```
#include <iostream.h>
#include <strstrea.h>

const int dim = 5;
class pct
  {
     int x, y;
    public:
     pct(int i, int j)
```

```
        {
          if(i>dim)
            i = dim;
          if(i<0)
            i = 0;
          if(j>dim)
            j = dim;
          if(j<0)
            j=0;
          x=i;
          y=j;
        }
      friend ostream &operator<<(ostream &flux, pct obj);
  };
ostream &operator<<(ostream & flux, pct obj)
  {
    register int i, j;
    for(j=dim; j>=0; j--)
      {
        flux << j;
        if(j==obj.y)
          {
            for(i=0; i<obj.x; i++)
              flux << " ";
            flux << '*';
          }
        flux << endl;
      }
    for(i=0; i<=dim; i++)
      flux << " " << i;
    flux << endl;
    return flux;
  }
void main(void)
  {
    pct a(2,3), b(1,1);
    char sr[200];
    cout << " Afisarea cu cout:" << endl;
    cout << a << endl << b << endl << endl;
    ostrstream outs(sr, sizeof(sr));
    outs << a << b << ends;
    cout << "Afisarea utilizand formatarea in-RAM:" << endl;
    cout << sr;
  }
```

În programul *pct_tabl.cpp*, operatorul personalizat de inserare creează o grilă simplă. Apoi utilizează informaţia conţinută în obiectul *pct* pentru a desena singurul punct al grilei. Programul efectuează atât afişarea standard, cât şi cea în flux pentru a arăta că aceleaşi

operator de inserare lucrează la fel de corect cu ambele fluxuri. Atunci când compilați și executați programul *pct_tabl.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Afisarea cu cout:
5
4
3  *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1  *
0
 0 1 2 3 4 5
Afisarea utilizand formatarea in-RAM:
5
4
3  *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1  *
0
 0 1 2 3 4 5
C:\>
```

1036 UTILIZAREA OPERATORILOR DE EXTRAGERE PERSONALIZAȚI CU FLUXURILE MATRICE

C/C++

După cum ați învățat în secțiunea 1035, veți crea operatori personalizați de inserare pentru fluxurile dumneavoastră matrice în aceeași manieră în care ați creat operatorii personalizați de inserare pentru fluxurile bazate pe fișiere. Crearea unui operator de extragere al unui flux matrice este la fel de simplă. Puteți crea operatori de extragere pentru matricele flux așa cum i-ați creat pe cei atașați fluxurilor bazate pe fișiere. Următorul program, *pct_tabl2.cpp* mai adaugă un operator personalizat de extragere la programul *pct_tabl.cpp* detaliat în secțiunea 1035:

```
#include <iostream.h>
#include <strstrea.h>

const int dim = 5;
class pct
   {
```

```
    int x, y;
   public:
    pct(void);
    pct(int i, int j)
      {
       if(i>dim)
         i = dim;
       if(i<0)
         i = 0;
       if(j>dim)
         j = dim;
       if(j<0)
         j=0;
       x=i;
       y=j;
      }
    friend ostream &operator<<(ostream &flux, pct obj);
  };
ostream &operator<<(ostream &flux, pct obj)
  {
   register int i, j;
   for(j=dim; j>=0; j--)
     {
      flux << j;
      if(j==obj.y)
        {
         for(i=0; i<obj.x; i++)
           flux << " ";
         flux << '*';
        }
      flux << endl;
     }
   for(i=0; i<=dim; i++)
     flux << " " << i;
   flux << endl;
   return flux;
  }
pct::pct(void)
  {
   cout << "Dati valoarea lui x: ";
   cin >> this->x;
   cout << "\nDati valoarea lui y: ";
   cin >> this->y;
  }
void main(void)
  {
   pct a(2,3), b(1,1), c;
```

```
    char sr[200];
    ostrstream outs(sr, sizeof(sr));
    cout << "Afisarea cu cout:" << endl;
    cout << a << endl << b << endl << c << endl << endl;
    outs << a << b << c << ends;
    cout << "Afisarea utilizand formatarea in-RAM:" << endl;
    cout << sr;
    istrstream ins(sr);
  }
```

Operatorul personalizat de extragere vă permite în plus să introduceți în clasă valorile pe care doriți ca programul dumneavoastră să le plaseze în grilă. Atunci când compilați și executați programul *pct_tabl2.cpp*, el va afișa ambele grile afișate în secțiunea 1035 și o a treia grilă care conține valoarea introdusă de dumneavoastră.

1037 UTILIZAREA MATRICELOR DINAMICE CU FLUXURILE DE I/O

C/C++

În secțiunile precedente ați învățat cum să utilizați constructorul *ostrstream* pentru a crea un flux matrice. De fiecare dată când utilizați constructorul *ostrstream*, declarați punctul de pornire și dimensiunea matricei. Totuși, pe măsură ce programele dumneavoastră devin mai complexe și vă veți acomoda mai bine cu C++, veți crea deseori matrice dinamice și nu matrice care au dimensiuni prestabilite. Pentru a crea un flux de ieșire care utilizează o matrice dinamică trebuie să utilizați constructorul clasei *ostrstream* într-un mod ușor diferit, așa cum arătăm mai jos:

```
#include <iostream.h>
#include <strstrea.h>

ostrstream(void);
```

Atunci când utilizați în programele dumneavoastră constructorul *ostrstream* fără parametri, *ostrstream* va crea și va menține o matrice alocată în mod dinamic. Constructorul *ostrstream* nu returnează un pointer către matrice. În schimb, dumneavoastră veți utiliza o a doua funcție, *sr*, care „îngheață" matricea și returnează un pointer la ea. După ce „înghețați" o matrice dinamică, nu o veți putea utiliza din nou pentru ieșiri, ci va trebui să creați o nouă matrice. Următorul program, *din_out.cpp*, utilizează un flux matrice de ieșire alocată dinamic:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char * p;
    ostrstream outs;
    outs << "Jamsa's C/C++ ";
    outs << "Programmer's Bible ";
    outs << "are programe interesante." << ends;
    p = outs.sr();
```

```
    cout << p;
    delete p;
  }
```

Atunci când compilaţi şi executaţi programul *din_out.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Jamsa's C/C++ Programmer's Bible are programe interesante.
C:\>
```

UTILITATEA PENTRU FORMATAREA FLUXURILOR MATRICE

C/C++1038

După cum aţi învăţat, C++ are posibilitatea de a supraîncărca operatorii de extragere, manipulatorii şi aproape toate celelalte instrumente de formatare împreună cu un flux „viu" (cu alte cuvinte, bazat pe fişiere), ceea ce elimină mult din necesitatea de formatare bazată pe RAM (cu alte cuvinte, fluxuri matrice). Totuşi, există câteva motive bune pentru care puteţi opta să utilizaţi formatarea fluxurilor matrice în cadrul programelor dumneavoastră, detaliate în această secţiune.

Una dintre cele mai răspândite utilizări ale formatării bazate pe matrice este atunci când programele dumneavoastră trebuie să construiască un şir de caractere pe care programul îl va utiliza mai târziu în cadrul fie al unei biblioteci standard, fie al unei terţe funcţii. De exemplu, poate că veţi avea nevoie să creaţi un şir de caractere pe care funcţia de bibliotecă standard *strtok* îl va analiza. Altă modalitate în care programele dumneavoastră vor utiliza I/O bazate pe matrice este de a crea un editor de texte care efectuează operaţii de formatare complexe – operaţii ce ar dura un timp semnificativ mai îndelungat dacă ar fi efectuate cu un fişier şi nu cu o matrice.

În fine, deoarece Windows nu deţine funcţii standard pe care le puteţi utiliza pentru a formata ieşirile într-o fereastră, trebuie să formataţi toate ieşirile înainte de a le trimite către fereastră. Utilizarea fluxurilor matrice sub mediul Windows este deseori foarte utilă în crearea de aplicaţii utile şi atractive.

MANIPULATORUL ENDS

C/C++1039

În secţiunile precedente aţi creat un flux de ieşire tip şir de caractere şi aţi utilizat operatorul de inserare pentru a pune manipulatorul *ends* în flux. Manipulatorul *ends* va plasa un caracter *NULL* în flux, tot aşa cum manipulatorul *endl* va insera caracterul de linie nouă. Atunci când utilizaţi operatorul de inserare pentru a insera text într-un buffer tip şir de caractere, veţi utiliza manipulatorul *ends* în mod regulat. Următorul program, *ends.cpp* utilizează manipulatorul *ends* cu câteva fluxuri de ieşire tip şiruri de caractere:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
  {
    char titlu[64], editura[64], autor[64];
    ostrstream titlu_sr(titlu, sizeof(titlu));
    ostrstream pub_sr(editura, sizeof(editura));
```

```
    ostrstream autor_sr(autor, sizeof(autor));
    titlu_sr << "Jamsa's C/C++ Programmer's Bible" << ends;
    pub_sr << "Jamsa Press" << ends;
    autor_sr << "Jamsa & Klander" << ends;
    cout << "Carte: " << titlu << " Editura: " << editura <<
        " Autor: " << autor << endl;
}
```

Cu toate că șirurile de caractere se termină în mod normal cu o valoare *NULL*, atunci când lucrați cu fluxuri matrice trebuie să precizați explicit manipulatorul *ends* pentru a indica fluxului să se încheie. În caz contrar, fluxul va rămâne deschis până când programul dumneavoastră se va ocupa din nou de flux. Manipulatorul *ends* nu este absolut necesar în situații simple, ca aceea arătată în programul *ends.cpp*, dar poate fi utilă în manevrarea mai profundă a fluxurilor matrice pentru operații de ieșire mult mai complexe.

1040 INVOCAREA UNUI OBIECT DE CĂTRE ALTUL C/C++

Pe măsură ce programele dumneavoastră în C++ devin mai complexe, ele vor începe să utilizeze mai multe tipuri de obiecte și prin urmare vor fi cazuri când un obiect va utiliza pe un altul. De exemplu, următorul program, *doua_obj.cpp* creează două tipuri diferite de obiecte: un obiect care conține informații despre un cititor și un obiect ce conține informații despre o carte. Obiectul *Cititor* apelează obiectul *Carte* pentru a afișa informații despre cartea preferată a unui cititor, așa cum arătăm mai jos:

```
#include <iostream.h>
#include <string.h>

class Carte {
   public:
     Carte(char *titlu) { strcpy(Carte::titlu, titlu); } ;
     void arata_carte(void) { cout << titlu; };
   private:
     char titlu[64];
};
class Cititor {
   public:
     Cititor(char *nume) { strcpy(Cititor::nume, nume); };
     void arata_cititor(class Carte carte)
       {
         cout << "Cititor: " << nume << endl << "Carte: ";
         carte.arata_carte();
       };
   private:
     char nume[64];
};
void main(void)
   {
     Cititor cititor("Kris Jamsa");
```

```
    Carte carte_preferata("Compiler Internals");
    cititor.arata_cititor(carte_preferata);
}
```

Programul *doua_obj.cpp* creează mai întâi instanța *cititor* a obiectului *Cititor* și atașează numele lui Kris Jamsa membrului privat *nume*. Apoi, programul creează o instanță a obiectului *Carte* denumită *carte_preferata*. Atunci când programul apelează funcția membru *cititor.arata_cititor* împreună cu obiectul *carte_preferata* ca parametru al funcției membre, programul apelează funcția membru *Carte.arata_carte* pentru obiectul *carte_preferata*. Atunci când compilați și executați programul *doua_obj.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Cititor: Kris Jamsa
Carte: Compiler Internals
C:\>
```

INFORMAREA COMPILATORULUI DESPRE O CLASĂ

C/C++ 1041

Atunci când o clasă referențiază identificatorul pentru o a doua clasă pe care nu ați declarat-o încă în cadrul programului dumneavoastră, trebuie să îi indicați compilatorului că identificatorul corespunde unei clase pe care o veți declara ulterior în program. Pentru aceasta, puteți plasa o instrucțiune în programul dumneavoastră care conține cuvântul cheie *class* și numele clasei, ca mai jos:

```
class nume_clasa;
```

De exemplu, să presupunem că clasa *Carte* dorește să îi indice compilatorului că clasa *Cititor* este clasa friend. Dacă nu ați declarat deja clasa *Cititor*, puteți plasa următoarea instrucțiune în programul dumneavoastră, care spune compilatorului că veți defini codul sursă al clasei mai târziu:

```
class Cititor;
```

Secțiunea 1043 utilizează această tehnică pentru a informa compilatorul despre clasa *Cititor* înainte de a referenția clasa *Cititor* în cadrul clasei *Carte*.

SĂ RECAPITULĂM FUNCȚIILE FRIEND

C/C++ 1042

După cum ați învățat, puteți declara funcții *friend* în cadrul programelor dumneavoastră. Deseori veți utiliza funcții *friend* pentru a supraîncărca funcții membre și operatori din cadrul unei clase date. Totuși, după cum ați învățat, C++ vă permite de asemenea să specificați o clasă ca fiind de tip *friend* pentru o alta, ceea ce permite clasei friend să acceseze datele și metodele private ale clasei. Pe când funcțiile *friend* (care de asemenea au acces la datele și metodele private ale clasei) sunt de obicei suficiente în majoritatea cazurilor, uneori se va întâmpla ca o clasă dintr-un program să aibă nevoie de acces complet la obiectele altei clase însă nederivând din acea clasă. În asemenea cazuri, puteți utiliza o clasă *friend*. Secțiunile 1043 și 1044 detaliază câteva clase *friend* simple.

1043 DECLARAREA CLASEI CITITOR CA FRIEND



În secțiunile precedente ați creat și utilizat clasa *Carte*, care menține informații despre titlul, autorul, editura și prețul unei cărți. Următorul program utilizează cuvântul cheie *friend* pentru a specifica faptul că clasa *Cititor* este prietenă a clasei *Carte*. Prin urmare, obiectele de tipul *Cititor* pot accesa membrii privați ai unui obiect *Carte*. În programul *citcarte.cpp*, clasa *Cititor* accesează data membră privată *titlu*, ca mai jos:

```
#include <iostream.h>
#include <string.h>

class Cititor;          // Declaratia va urma
class Carte
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); } ;
      void arata_carte(void) { cout << titlu; };
      friend Cititor;
    private:
      char titlu[64];
  };
class Cititor {
  public:
    Cititor(char *nume) { strcpy(Cititor::nume, nume); };
    void arata_cititor(class Carte carte)
      {
        cout << "Cititor: " << nume << ' ' << "Carte: "
           << carte.titlu;
      };
  private:
    char nume[64];
  };
void main(void)
  {
    Cititor cititor("Kris Jamsa");
    Carte carte_preferata("Compiler Internals");
    cititor.arata_cititor(carte_preferata);
  }
```

După cum puteți vedea, în cadrul obiectului *Carte*, următoarea instrucțiune spune compilatorului că obiectul *Cititor* este un *prieten (friend)*, ceea ce permite obiectului *Cititor* să acceseze membrii privați ai obiectului *Carte*:

```
friend Cititor
```

ALT EXEMPLU DE CLASĂ FRIEND

C/C++ 1044

După cum ați învățat în secțiunile precedente, puteți declara o întreagă clasă ca fiind *friend* pentru o altă clasă. Acordând celei de-a doua clase statutul de *friend* îi permite acesteia să acceseze toți membrii primei clase, aceasta incluzând aspecte ca numele de tipuri și constantele enumerate. Următorul program, *fac_schimb.cpp*, utilizează o clasă *friend* pentru a accesa o enumerare privată:

```
#include <iostream.h>

class cantit;
class monezi
  {
    enum unitati {penny, nickel, dime, quarter, half_dollar};
    friend cantit;
  };
class cantit
  {
     monezi::unitati moneda;
    public:
     void setm(void);
     int getm(void);
  } obiect;
void cantit::setm(void)
  {
    moneda = monezi::dime;
  }
int cantit::getm(void)
  {
    return moneda;
  }
void main(void)
  {
    obiect.setm();
    cout << obiect.getm();
  }
```

În programul *fac_schimb.cpp*, se declară enumerativ tipurile de unități monetare în cadrul primei clase (*monezi*), apoi se declară a doua clasă (*cantit*) pentru a păstra un anumit număr sau tip de *unitati*. Deoarece programul declară clasa *cantit* ca fiind *friend* pentru clasa *monezi*, clasa *cantit* are acces la membrul privat *unitati*.

ELIMINAREA NECESITĂȚII INSTRUCȚIUNII CLASS NUME_CLASA

C/C++ 1045

După cum ați învățat, atunci când o clasă referențiază un identificator pentru o clasă pe care programul dumneavoastră nu a declarat-o încă, puteți informa compilatorul despre faptul că identificatorul corespunde unei clase utilizând o instrucțiune similară următoarei:

```
class nume_clasa;
```

Incluzând o astfel de instrucțiune, clasa se poate referi la cea de-a doua clasă utilizând numai identificatorul clasei, ca mai jos:

```
friend nume_clasa;
```

Dacă plasați instrucțiunea *class* între *friend* și *nume_clasa*, eliminați nevoia declarației ulterioare:

```
friend class nume_clasa;
```

Următoarea declarație a clasei *Carte* utilizează ultima tehnică pentru a informa compilatorul de faptul că clasa *Cititor* este clasă prietenă:

```
class Carte
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); } ;
      void arata_carte(void) { cout << titlu; };
      friend class Cititor;
    private:
      char titlu[64];
  };
```

1046 RESTRICȚIONAREA ACCESULUI UNEI CLASE FRIEND

C/C++

După cum ați învățat, C++ vă permite să specificați faptul că un obiect este *friend* pentru un altul, ceea ce permite obiectului *friend* să aibă acces la membrii privați ai obiectului. Pentru a controla mai bine accesul obiectului *friend* a membrii privați ai obiectului, limbajul C++ vă permite să specificați acele metode din clasa friend care pot accesa membrii privați. Celelalte metode ale obiectului *friend* nu au acces la membrii privați. De exemplu, presupunând că numai membrul *arata_carte* din clasa *Cititor* necesită acces la membrii privați ai clasei *Carte*. În cadrul clasei *Carte*, puteți plasa următoarea instrucțiune:

```
friend Cititor::arata_carte(void);
```

Următorul program, *friend2.cpp*, utilizează *formatul* restricționat al instrucțiunii *friend* pentru a limita accesul clasei *Cititor* la obiectul *Carte*, astfel încât numai funcția *arata_cititor* să poată accesa obiectul *Carte*:

```
#include <iostream.h>
#include <string.h>

class Cititor
  {
    public:
      Cititor(char *nume) { strcpy(Cititor::nume, nume); };
      void arata_cititor(class Carte carte);
```

```
    private:
      char nume[64];
  };
class Carte
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); } ;
      void arata_carte(void) { cout << titlu; };
      friend Cititor::arata_cititor(Carte carte);
    private:
      char titlu[64];
  };
void Cititor::arata_cititor(class Carte carte)
  { cout << "Cititor: " << nume << ' ' << "Carte: "
     << carte.titlu; };
void main(void)
  {
    Cititor cititor("Kris Jamsa");
    Carte carte_preferata("Compiler Internals");
    cititor.arata_cititor(carte_preferata);
  }
```

Cu toate că programul *friend2.cpp* va afișa același rezultat ca și programul *doua_obj.cpp* descris în secțiunea 1040, el este un program mai atent deoarece el oferă clasei *Cititor* numai un acces limitat asupra clasei *Carte*. După cum ați învățat, unul din cele mai semnificative pericole ale claselor friend este accesul lor nelimitat la membrii interni ai claselor pentru care sunt *friend*. Limitarea accesului claselor *friend* este un pas important în protejarea împotriva accesului nelimitat.

CONFLICTELE DE NUME ȘI CLASELE FRIEND

C/C++1047

Atunci când clasele dumneavoastră utilizează tipul *friend* pentru a accesa membrii altei clase, uneori se va întâmpla ca numele membrilor să creeze un conflict între cele două clase. Atunci când apare un astfel de conflict, programul utilizează membrul clasei curente. Următorul program, *memconfl.cpp*, ilustrează un conflict de nume între clase *friend*:

```
#include <iostream.h>
#include <string.h>
class Carte
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); } ;
      void arata_carte(void) { cout << titlu; };
      friend class Cititor;
    private:
      char titlu[64];
```

```
   };
class Cititor
   {
     public:
       Cititor(char *nume) { strcpy(Cititor::nume, nume); };
       void arata_cititor(class Carte carte) {
           cout << "Cititor: " << nume << ' ' << "Carte: " <<
                       carte.titlu; };
       void arata_carte(void) { cout << "Cititorul cartii este
          " << nume << endl; } ;
     private:
       char nume[64];
   };
void main(void)
   {
     Cititor cititor("Kris Jamsa");
     Carte carte_preferata("Compiler Internals");
     cititor.arata_carte();
     cititor.arata_cititor(carte_preferata);
   }
```

După cum puteți vedea, ambele clase utilizează numele membrului *arata_carte*. Atunci când compilați și executați programul *memconfl.cpp*, el va utiliza membrul clasei *Cititor*, ca mai jos:

```
Cititorul cartii este Kris Jamsa
Cititor: Kris Jamsa Carte: Compiler Internals
C:\>
```

1048 MOȘTENIREA ÎN C++

C/C++

După cum ați învățat pe scurt în secțiunile precedente, C++ acceptă *moștenirea*, ceea ce vă permite să derivați o clasă nouă dintr-o clasă existentă sau o clasă de bază. Atunci când derivați o clasă din alta în C++, veți folosi următorul format:

```
class clasa_derivata: public clasa_de_baza {
   public:
     // membrii publici ai clasei derivate
   private:
     //membrii privati ai clasei derivate
};
```

Ca un exemplu de clase derivate, următorul program, *constr_bibl.cpp* creează o clasă de bază denumită *Carte* și apoi derivează din această clasă de bază o clasă denumită *FisaBiblioteca*:

```
#include <iostream.h>
#include <string.h>

class Carte
```

```
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); };
      void arata_titlu(void) { cout << titlu << endl; };
    private:
      char titlu[64];
  };

class FisaBiblioteca : public Carte
  {
    public:
      FisaBiblioteca(char *titlu, char *autor, char *editura) :
        Carte(titlu)
        {
          strcpy(FisaBiblioteca::autor, autor);
          strcpy(FisaBiblioteca::editura, editura);
        };
      void arata_Biblioteca(void)
        {
          arata_titlu();
          cout << autor << ' ' << editura;
        };
    private:
      char autor[64];
      char editura[64];
  };

void main(void)
  {
    FisaBiblioteca fisa("Jamsa's C/C++ Programmer's Bible",
                        "Jamsa & Klander", "Jamsa Press");
    fisa.arata_Biblioteca();
  }
```

Conceptul de *moștenire* este extrem de important în programarea orientată pe obiect. Posibilitatea de a crea o ierarhie de clase, de la cea mai generală către cea mai specifică, vă permite să controlați mai bine programele dumneavoastră. Mai mult, aceasta face ca programele dumneavoastră să fie mai ușor de înțeles pentru alți cititori și să fie mai ușor de dezvoltat. Pe când clasele sunt un instrument util de programare în totalitatea lor, moștenirea și proprietățile asociate ei conferă adevărata forță a programării în C++. Veți învăța pe larg despre moștenire în următoarele secțiuni.

CLASELE DE BAZĂ ȘI CELE DERIVATE C/C++ 1049

Un element fundamental al conceptului de moștenire este relația dintre clasele de bază și cele derivate. Atunci când construiți o clasă utilizând o clasă existentă, noua clasă moștenește caracteristicile clasei deja existente. Caracteristicile clasei existente cuprind datele și metodele, cât și accesul lor (public sau privat). După cum ați citit prin cărți și reviste despre programarea orientată pe obiecte, veți întâlni termenii de *clasă de bază* și *clasă*

derivată. Clasa de bază este clasa originară ale cărei caracteristici sunt moștenite de cealaltă clasă. Clasa derivată este clasa pe care programul dumneavoastră o creează din clasa de bază. Mai multe clase diferite pot utiliza aceeași clasă de bază. Și invers, puteți construi o clasă derivată pornind de la câteva clase de bază distincte. Figura 1049 prezintă o derivare simplă dintr-o clasă de bază.

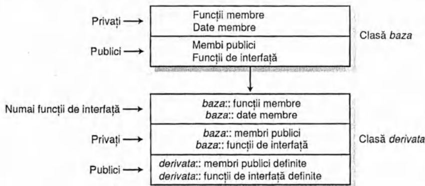


Figura 1049 *Un exemplu simplu de moștenire.*

1050 DERIVAREA UNEI CLASE

C/C++

În secțiunea 1048, programul *constr_bibl.cpp* a derivat clasa *FisaBiblioteca* utilizând clasa de bază *Carte*. Prima linie a declarației clasei *FisaBiblioteca* informează compilatorul că *FisaBiblioteca* este o clasă derivată care utilizează clasa de bază *Carte*, ca mai jos:

```
class FisaBiblioteca : public Carte {
```

Restul declarației clasei este foarte asemănător celui cu care ați mai lucrat, cu excepția funcției constructor. După cum puteți vedea, declarația imediat următoare, a constructorului *FisaBiblioteca* este o apelare a constructorului *Carte*:

```
FisaBiblioteca(char *titlu, char *autor, char *editura):
  Carte(titlu)
  {
    strcpy(FisaBiblioteca::autor, autor);
    strcpy(FisaBiblioteca::editura, editura);
  };
```

Atunci când programul creează o nouă instanță a clasei *FisaBiblioteca* (și prin urmare apelează constructorul *FisaBiblioteca*), constructorul *FisaBiblioteca* va apela mai întâi constructorul clasei de bază (pentru *Carte*). Dacă nu specificați constructorul clasei de bază după constructorul *FisaBiblioteca*, ca în fragementul de cod precedent, compilatorul va genera o eroare de sintaxă. După cum veți învăța în următoarele secțiuni, compilatorul va genera erori de sintaxă întrucât constructorul clasei *Carte* necesită un singur parametru. În cazul în care constructorul clasei *Carte* ar fi lipsit de parametri, compilatorul nu vă impune să specificați clasa de bază, așa cum am arătat în fragmentul de cod precedent.

CONSTRUCTORII DE BAZĂ ȘI CEI DERIVAȚI

C/C++ 1051

Atunci când derivați o clasă dintr-o clasă de bază care are o funcție constructor, trebuie să invocați constructorul clasei de bază din interiorul constructorului clasei derivate. Următorul program, *bazaderi.cpp*, derivează o clasă denumită *Derivata* dintr-o clasă de bază *Baza*. În cadrul funcției constructor a clasei *Derivata*, codul apelează funcția constructor a clasei *Baza*, așa cum arătăm mai jos:

```
#include <iostream.h>

class Baza
  {
    public:
      Baza(void) { cout << "Constructorul clasei Baza\n"; };
  };

class Derivata:Baza
  {
    public:
      Derivata(void): Baza()
        { cout << " Constructorul clasei Derivata\n"; };
  };

void main(void)
  {
    Derivata obiect;
  }
```

De fiecare dată când creați o instanță a unei clase derivate, programul va executa atât constructorul acelei clase cât și pe cel al clasei de bază. Atunci când compilați și executați programul *bazaderi.cpp*, pe ecranul dumneavoastră va apărea următorul rezultat:

```
Constructorul clasei de baza
Constructorul clasei derivate
C:\>
```

După cum puteți vedea, constructorul clasei de bază se execută înaintea celui al clasei derivate.

UTILIZAREA MEMBRILOR DE TIPUL PROTECTED

C/C++ 1052

După cum ați învățat, C++ permite programelor dumneavoastră să declare membri de tip *protected* ai claselor dumneavoastră, care sunt complet accesibili claselor pe care programele dumneavoastră le vor deriva din aceasta. Următorul program, *protect.cpp*, ilustrează modul de utilizare al membrilor protejati. În programul *protect.cpp*, clasa *Carte* definește câțiva membrii protejați. Apoi, programul derivează clasa *FisaBiblioteca*, din clasa de bază *Carte*. Rețineți, clasa derivată, *FisaBiblioteca*, poate accesa membrii protejați din cadrul clasei Carte, ceea ce permite executarea cu succes a apelării funcției *arata_pret* din cadrul clasei *FisaBiblioteca*, precum și a modificării membrului *cost*. Următorul cod este implementarea programului *protect.cpp*:

```
#include <iostream.h>
#include <string.h>

class Carte
  {
    public:
      Carte(char *titlu) { strcpy(Carte::titlu, titlu); };
      void arata_titlu(void) { cout << titlu << endl; };
    protected:
      float cost;
      void arata_cost(void) { cout << cost << endl; };
    private:
      char titlu[64];
  };

class FisaBiblioteca : public Carte
  {
    public:
      FisaBiblioteca(char *titlu, char *autor, char *editura) :
        Carte(titlu)
        {
          strcpy(FisaBiblioteca::autor, autor);
          strcpy(FisaBiblioteca::editura, editura);
          cost = 49.95;
        };
      void arata_Biblioteca(void)
        {
          arata_titlu();
          arata_cost();
          cout << autor << ' ' << editura;
        };
    private:
      char autor[64];
      char editura[64];
  };

void main(void)
  {
    FisaBiblioteca fisa("Jamsa's C/C++ Programmer's Bible",
                        "Jamsa & Klander", "Jamsa Press");
    fisa.arata_Biblioteca();
  }
```

Atunci când compilați și executați programul *protect.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Jamsa's C/C++ Programmer's Bible
49.95
Jamsa & Klander Jamsa Press
```

După cum puteți vedea, instrucțiunile clasei *FisaBiblioteca* au acces deplin la membrii protejați ai clasei de bază *Carte*.

CÂND UTILIZĂM MEMBRII DE TIP PROTECTED

C/C++1053

După cum ați învățat, clasele derivate pot accesa membrii protejați ai clasei lor de bază. Când creați clasele, trebuie să decideți care membri să fie făcuți publici, privați sau protejați. Ca regulă, trebuie să creați fiecare clasă cu intenția ca o clasă derivată să o poată utiliza ulterior. Dacă nu veți utiliza niciodată clasa ca o clasă de bază, membrii protejați vor fi în principal privați. Dacă ulterior vă veți decide să utilizați clasa ca pe o clasă de bază, predeteminarea membrilor de tip *protected* vă va ajuta să economisiți timp de programare.

RECAPITULAREA MOȘTENIRII PUBLICE ȘI PRIVATE

C/C++1054

După cum ați învățat, programele dumneavoastră pot deriva o a doua clasă dintr-o primă clasă. Până acum, ați derivat cea de-a doua clasă din prima clasă ca pe o derivare *publică*, ceea ce înseamnă că obiectele clasei derivate pot accesa membri ai clasei de bază. Pentru a înțelege mai bine aceasta, analizați programul *bazader2.cpp*, arătat mai jos:

```
#include <iostream.h>

class Baza
  {
   public:
     Baza(void) { cout << "Constructorul clasei Baza\n"; };
     int data;
  };

class Derivata:public Baza
  {
   public:
     Derivata(void): Baza() { cout << "Constructorul clasei
        Derivata\n"; };
  };

void main(void)
  {
   Derivata obiect;
   obiect.data = 5;
   cout << obiect.data << endl;
  }
```

Atunci când compilați și executați programul *bazader2.cpp*, *obiectul* pe care l-ați creat din clasa *Derivata* poate accesa direct membrul *data* al clasei *Baza*. După cum știți, acest tip de acces direct încalcă regulile încapsulării și de accea trebuie evitat. Ca o alternativă. Programele dumneavoastră pot moșteni clasa *Baza* în mod *privat*, ceea ce are ca efect transformarea tuturor membrilor clasei *Baza* în membri privați, așa cum arătăm în programul

```
#include <iostream.h>
class Baza
  {
   public:
    Baza(void) { cout << "Constructorul clasei Baza\n"; };
    int data;
  };
class Derivata: private Baza
  {
   public:
    Derivata(void): Baza() { cout << "Constructorul clasei
      Derivata\n"; };
  };
void main(void)
  {
   Derivata obiect;
   obiect.data = 5;
   cout << obiect.data << endl;
  }
```

Atunci când programul *bazader3.cpp* încearcă să acceseze membrul *data* al clasei *Baza*, compilatorul eșuează și returnează o eroare. Totuși, clasa *Derivata* poate încă accesa constructorul clasei de bază, din cadrul obiectului. În plus, o funcție de interfață ar permite programului dumneavoastră să acceseze și membrul *data*. În secțiunea 1055 veți învăța despre cea de-a treia metodă de derivare a claselor de bază, cea *protejată*.

1055 MOȘTENIREA PROTEJATĂ A CLASEI DE BAZĂ C/C++

După cum ați învățat, atunci când creați clase în programele dumneavoastră, puteți utiliza cuvântul cheie *protected* pentru a preveni accesul altor părți ale programului la anumiți membri ai clasei, dar lăsând în continuare membrii clasei disponibili claselor moștenitoare. De asemenea puteți moșteni o întreagă clasă ca *protected*. Atunci când moșteniți o întreagă clasă ca *protected*, toți membrii publici și portejați ai clasei de bază devin membri protejați ai clasei derivate. Pentru a înțelege mai bine modul în care clasele derivate moștenesc membrii publici și protejați, analizați următorul program, *prot_baz.cpp*:

```
#include <iostream.h>
class baza
  {
   protected:
    int i, j;
   public:
    void setij(int a, int b)
      {
       i=a;
       j=b;
```

```
    }
    void arataij(void) {cout << i << " " << j << endl;}
  };
class derivata : protected baza
  {
    private:
     int k;
    public:
     void setk(void)
       {
        setij(10,12);
        k = i * j;
       }
     void arataall(void)
       {
        cout << k << " ";
        arataij();
       }
  };
void main(void)
  {
    derivata obiect;
    // obiect.setij(2,3); Aceasta este o comanda ilegala,
    // deoarece setij este un membru protejat al clasei derivata.
    // obiect.arataij(); Aceasta este de asemenea o comanda
    // ilegala.
    obiect.setk();
    obiect.arataall();
  }
```

După cum puteți vedea, clasa *derivata* folosește cuvântul cheie *protected* pentru a deriva clasa *baza*. Acest lucru are drept consecință transformarea tuturor membrilor clasei *baza* în membri *privați* în cadrul clasei *derivata*. Dacă apoi programul încearcă să acceseze un membru descendent *privat* (cum ar fi *setij* sau *arataij*), compilatorul va returna o eroare deoarece membrul este încapsulat. Codul programului notează faptul că ambele funcții *obiect.setij(2, 3)* și *obiect.arataij()* sunt apeluri de funcții ilegale, deoarece membrii sunt protejați în cadrul clasei derivate, ceea ce înseamnă că singurele obiecte ce pot accesa funcțiile pe care clasa *derivata* le moștenește de la clasa *baza* sunt funcții de interfață sau clase pe care programul dumneavoastră le derivează din clasa *derivata*. Atunci când compilați și executați programul *prot_baz.cpp*, ecranul dumneavoastră va afișa următoarele:

```
120 10 12
C:\>
```

MOȘTENIREA MULTIPLĂ

C/C++1056

După cum ați învățat, moștenirea este caracteristica unei clase de a moșteni caracteristicile unei alte clase. Moștenirea multiplă este abilitatea unei clase de a moșteni caracteristicile a

mai multor clase de bază. C++ acceptă moștenirea multiplă. Atunci când o clasă derivată moștenește caracteristici din mai multe clase de bază, pur și simplu separați numele claselor de bază utilizând virgule, ca mai jos:

```
class derivata: public clasa_baza1, public clasa_baza2
  {
    public:
      // Membrii publici ai clasei derivate
    private:
      // Membrii privati ai clasei derivate
  }
```

În mod asemănător, când declarați apoi funcția constructor pentru clasa derivată, trebuie să apelați funcțiile constructor pentru fiecare din clasele de bază. Secțiunea 1057 ilustrează un model simplu de moștenire multiplă.

1057 O MOȘTENIRE MULTIPLĂ SIMPLĂ

C/C++

După cum ați învățat, moștenirea multiplă este caracteristica unei clase derivate de a moșteni caracteristicile a două sau mai multe clase de bază. Pentru mulți programatori, înțelegerea modului în care o clasă unică poate moșteni caracteristicile altor două clase poate fi dificilă. Următorul program, *simpmult.cpp*, ilustrează cum creează moștenirea multiplă o clasă denumită *Carte*, care moștenește clasele de bază *Pagina* și *Coperta*:

```
#include <iostream.h>
#include <string.h>

class Coperta
  {
    public:
      Coperta(char *titlu) { strcpy(Coperta::titlu, titlu); };
    protected:
      char titlu[256];
  };

class Pagina
  {
    public:
      Pagina(int linii = 55) { Pagina::linii = linii; };
    protected:
      int linii;
      char *text;
  };

class Carte: public Coperta, public Pagina
  {
    public:
      Carte(char *autor, char *titlu, float cost): Coperta(titlu),
           Pagina(60)
        {
```

```
         strcpy(Carte::autor, autor);
         strcpy(Carte::titlu, titlu);
         Carte::cost = cost;
       };
     void arata_carte(void)
       {
         cout << titlu << endl;
         cout << autor << '\t' << cost;
       };
   private:
     char autor[256];
     float cost;
 };
void main(void)
 {
   Carte text("Jamsa and Klander", "Jamsa's C/C++ Programmer's
     Bible", 49.95);
   text.arata_carte();
 }
```

Programul *simpmult.cpp* definește două clase, *Coperta* și *Pagina*. Apoi, programul derivează o a treia clasă, *Carte* din amândouă clasele originare. Constructorul *Carte* transmite parametrii către constructorii celor două clase de bază. După ce programul creează o instanță a obiectului *Carte*, el generează afișarea informațiilor despre obiect. Funcția *arata_carte* afișează valori, câte una din fiecare clasă: *Coperta*, *Pagina* și *Carte*, așa cum arătăm mai jos:

```
Jamsa's C/C++ Programmer's Bible
Jamsa and Klander
 49.95
C:\>
```

ORDINEA CONSTRUCTORILOR ȘI CLASELE DE BAZĂ

C/C++ 1058

Moștenirea multiplă este caracteristica unei clase de a moșteni caracteristicile a mai multor clase de bază. Atunci când utilizați moștenirea multiplă pentru a deriva o clasă, clasa derivată trebuie să apeleze funcțiile constructor ale fiecăreia dintre clasele de bază. Ordinea apelării constructorilor depinde de ordinea în care clasa derivată a specificat clasele de bază. Cu alte cuvinte, când clasa derivată, *Derivata*, specifică clasele ei de bază ca fiind *Unu* și *Doi*, programul va apela constructorii în următoarea ordine: *Unu*, apoi *Doi*, apoi *Derivata*, ceea ce înseamnă că va fi un cod similar cu următorul:

```
derivata(void) : Unu(),Doi(int i);
```

Pentru a înțelege mai bine ordinea în care programele dumneavoastră trebuie să specifice parametrii constructorilor, analizați următorul program, *multinv.cpp*, care ilustrează ordinea apelării constructorilor atunci când derivați o clasă din trei clase de bază:

```
#include <iostream.h>
class Unu
   {
    public:
      Unu(void) { cout << "Constructorul clasei Unu\n"; };
   };
class Doi
   {
    public:
      Doi(void) { cout << "Constructorul clasei Doi\n"; };
   };
class Trei
   {
    public:
      Trei(void) { cout << "Constructorul clasei Trei\n"; };
   };
class Derivata: public Unu, public Trei, public Doi
   {
    public:
      Derivata(void) : Unu(), Doi(), Trei()
        { cout << "Apelam constructorul clasei Derivata\n"; };
   };
void main(void)
   {
    Derivata clasa_mea;
   }
```

Atunci când compilați și executați programul *multinv.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Constructorul clasei Unu
Constructorul clasei Doi
Constructorul clasei Trei
C:\>
```

După cum puteți vedea, programul apelează funcțiile constructor în aceeași ordine ca cea a numelor claselor de bază din antetul clasei:

```
class Derivata: public Unu, public Doi, public Trei
```

1059 DECLARAREA UNEI CLASE DE BAZĂ CA PRIVATĂ

C/C++

După cum ați învățat, programele dumneavoastră pot moșteni o singură clasă utilizând cuvintele cheie *public*, *private* sau *protected*. Atunci când derivați o clasă, puteți precede numele clasei de bază cu *private* sau *public*. Atunci când programele dumneavoastră utilizează cuvântul cheie *public* pentru a deriva o clasă de bază, programul dumneavoastră

poate utiliza clasa derivată pentru a accesa în mod direct membrii publici ai clasei de bază. Însă atunci când utilizați cuvântul cheie *private*, programul dumneavoastră poate accesa membrii clasei de bază doar prin intermediul membrilor clasei derivate. În fine, atunci când programele dumneavoastră utilizează cuvântul cheie *protected*, clasa derivată moștenește toți membrii *publici* din cadrul clasei moștenite, ca membri *protejați*, ceea ce permite clasei derivate accesul la membrii clasei de bază, dar permite și altor clase să deriveze membrii protejați. Însă, fiecare dintre secțiunile precedente care au ilustrat moștenirea multiplă a utilizat cuvântul cheie *public* în fața numelor clasei de bază, ca mai jos:

```
Class Derivata: public Unu, public Trei, public Doi {
```

Când apare această situație, programele dumneavoastră pot utiliza cuvintele cheie *public*, *private* și *protected* când programele efectuează moșteniri multiple, la fel cum ar face în cazul moștenirii simple. Următorul program, *privmult.cpp*, derivează o clasă din două clase de bază ale căror nume sunt precedate de cuvântul cheie *private* și dintr-o clasă de bază cu nume precedat de cuvântul cheie *public*:

```
#include <iostream.h>
class Unu
  {
    public:
      Unu(void)
        {
          cout << "Constructor pentru Unu\n";
          unu = 1;
        };
      int unu;
  };
class Doi
  {
    public:
      Doi(void)
        {
          cout << "Constructor pentru Doi\n";
          doi = 2;
        };
      int doi;
  };
class Trei
  {
    public:
      Trei(void)
        {
          cout << "Constructor pentru Trei\n";
          trei = 3;
        };
      int trei;
  };
```

```
class Derivata: private Unu, private Trei, public Doi
  {
   public:
     Derivata(void) : Unu(), Doi(), Trei()
       { cout << "Apelat constructor Derivata\n"; };
     void arata_valoare(void) { cout << unu << doi << trei <<
       endl; };
  };

void main(void)
  {
   Derivata clasa_mea;
   clasa_mea.arata_valoare();
   cout << clasa_mea.doi;
  }
```

Deoarece clasa derivată declară clasa de bază Doi cu public, programul *privmult.cpp* poate accesa direct membrul doi fără a fi nevoie să utilizeze funcțiile de interfață. Dar, programul nu poate accesa direct valorile unu și trei deoarece clasa derivată declară clasele de bază Unu și Trei ca private. În schimb, programul trebuie să folosească o funcție membră, cum este *arata_valoare*, pentru a afișa valorile unu și trei.

1060 FUNCȚIILE DESTRUCTOR ȘI MOȘTENIREA MULTIPLĂ

C/C++

După cum ați învățat, atunci când derivați o clasă dintr-o clasă de bază, C++ apelează constructorul clasei de bază după ce apelează funcția constructor a clasei derivate. În cazul funcțiilor destructor, însă, se întâmplă exact invers: C++ apelează destructorul clasei derivate și apoi apelează destructorul fiecăreia dintre clasele de bază. Următorul program, *destmult.cpp*, ilustrează secvența apelării funcțiilor destructor ale clasei derivate și ale claselor de bază:

```
#include <iostream.h>

class Unu
  {
   public:
     Unu(void) { cout << "Constructor pentru Unu\n"; };
     ~Unu(void) { cout << "Destructor pentru Unu\n"; };
  };

class Doi
  {
   public:
     Doi(void) { cout << "Constructor pentru Doi\n"; };
     ~Doi(void) { cout << "Destructor pentru Doi\n"; };
  };

class Trei
  {
   public:
```

```
    Trei(void) { cout << "Constructor pentru Trei\n"; };
    ~Trei(void) { cout << "Destructor pentru Trei\n"; };
 };

class Derivata: public Unu, public Doi, public Trei
 {
   public:
    Derivata(void) : Unu(), Doi(), Trei()
      { cout << "Apelat constructor Derivata\n"; };
    ~Derivata(void)
      { cout << "Apelat destructor Derivata\n"; };
 };

void main(void)
 {
   Derivata clasa_mea;
 }
```

Atunci când compilați și executați programul *destmult.cpp*, ecranul dumneavoastră va afișa următoarea ieșire:

```
Constructor pentru Unu
Constructor pentru Doi
Constructor pentru Trei
Apelat constructor Derivata
Apelat destructor Derivata
Destructor pentru Trei
Destructor pentru Doi
Destructor pentru Unu
C:\>
```

După cum vedeți, C++ invocă funcțiile destructor în ordinea opusă invocării funcțiilor constructor.

CONFLICTELE DE NUME ÎNTRE CLASELE DERIVATE ȘI CELE DE BAZĂ

C/C++1061

Atunci când derivați o nouă clasă utilizând una sau mai multe clase de bază, este posibil ca numele unui membru din clasa derivată să fie identic cu numele unui membru din una sau mai multe clase de bază. Atunci când apare un astfel de conflict, C++ utilizează numele membrului din clasa derivată. Următorul program, *conflict.cpp*, ilustrează un conflict de nume între numele unui membru al unei clase de bază și numele unui membru al clasei derivate:

```
#include <iostream.h>

class Baza
 {
   public:
    void afisare(void) { cout << "Aceasta este clasa Baza"
      << endl; };
 };
```

```
class Derivata: public Baza
   {
     public:
       void afisare(void) { cout << "Aceasta este clasa Derivata"
          << endl; };
   };

void main(void)
   {
     Derivata clasa_mea;
     clasa_mea.afisare();
   }
```

Atunci când compilați și executați programul *conflict.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este clasa derivata
C:\>
```

Deoarece funcția membru *afisare* este parte atât a clasei *Baza*, cât și a clasei *Derivata*, există un potențial conflict. Totuși, deoarece variabila *clasa_mea* este obiect al clasei *Derivata*, compilatorul va utiliza versiunea din clasa *Derivata* a funcției *afisare*.

1062 REZOLVAREA CONFLICTELOR DE NUME DINTRE CLASELE DE BAZĂ ȘI CELE DERIVATE

C/C++

După cum ați învățat în secțiunea 1061, atunci când numele unui membru al clasei derivate intră în conflict cu numele unui membru al unei clase de bază, C++ va folosi numele membrului clasei derivate. Însă, uneori poate că programele dumneavoastră vor trebui să acceseze membrul clasei de bază. Pentru a face aceasta, programul dumneavoastră poate să folosescă operatorul de rezoluție globală (::). Următorul program, *reznume.cpp*, utilizează operatorul de rezoluție globală pentru a accesa membrul clasei de bază *afisare*:

```
#include <iostream.h>

class Baza
   {
     public:
       void afisare(void) { cout << "Aceasta este clasa Baza"
          << endl; };
   };

class Derivata: public Baza
   {
     public:
       void afisare(void) { cout << "Aceasta este clasa Derivata"
          << endl; };
   };

void main(void)
   {
```

```
    Derivata clasa_mea;
    clasa_mea.afisare();
    clasa_mea.Baza::afisare();
}
```

Atunci când compilați și executați programul *reznume.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este clasa Derivata
Aceasta este clasa Baza
C:\>
```

După cum ați văzut în secțiunea 1061, referința simplă la funcția *afisare* a apelat funcția clasei derivate. Însă, deoarece a doua instrucțiune precede numele funcției cu operatorul de rezoluție globală și numele clasei *Baza*, a doua instrucțiune apelează funcția membru *afisare* din cadrul clasei *Baza*.

CÂND EXECUTĂ CLASELE MOȘTENITORI CONSTRUCTORII

C/C++1063

După cum ați învățat, funcția constructor a clasei derivate va apela întotdeauna funcția constructor a clasei din care a fost derivată. Totuși, clasele derivate pot de asemenea să efectueze propriile prelucrări în cadrul funcției constructor a clasei derivate. De fapt, atunci când programele dumneavoastră derivează clase, fiecare instanță a clasei va apela funcția constructor a fiecărei clase din arborele ierarhic superior clasei respective, înainte de a efectua funcția constructor a clasei derivate. De exemplu, să presupunem că clasa dumneavoastră *derivata5* este derivată din alte patru clase derivate și o clasă de bază, *baza*. Dacă fiecare constructor de clasă afișează câte un mesaj, atunci când creați o instanță a clasei *derivata5*, constructorii dumneavoastră vor afișa următoarele mesaje:

```
Construieste clasa baza
Construieste clasa derivata1
Construieste clasa derivata2
Construieste clasa derivata3
Construieste clasa derivata4
Construieste clasa derivata5
```

După cum ați învățat, programul execută funcțiile destructor în ordine inversă față de funcțiile constructor, astfel încât acestea vor afișa o serie de mesaje similare cu următoarele:

```
Distruge clasa derivata5
Distruge clasa derivata4
Distruge clasa derivata3
Distruge clasa derivata2
Distruge clasa derivata1
Distruge clasa baza
```

UN EXEMPLU DE CONSTRUCTOR AL UNEI CLASE MOȘTENITE

C/C++1064

După cum ați învățat, programele în C++ apelează constructorul și destructorul fiecărei clase superioare în arborele ierarhic unei clase derivate. Pentru a înțelege mai bine modul în care

C++ apelează funcțiile constructor și destructor pentru clasele derivate, analizați programul *cons_des.cpp*. Programul, *cons_des.cpp* construiește și distruge un singur obiect de tipul *derivata2* și apelează constructorii și destructorii claselor superioare în arborele ierarhic clasei *derivata2*, ca mai jos:

```
#include <iostream.h>

class baza
  {
    public:
      baza(void) {cout << "Construieste clasa baza.\n";}
      ~baza(void) {cout << "Distruge clasa baza.\n";}
  };

class derivata1 : public baza
  {
    public:
      derivata1(void) {cout << "Construieste clasa
        derivata1.\n";}
      ~derivata1(void) {cout << "Distruge clasa derivata1.\n";}
  };

class derivata2 : public derivata1
  {
    public:
      derivata2(void) {cout << "Construieste clasa
        derivata2.\n";}
      ~derivata2(void) {cout << "Distruge clasa derivata2.\n";}
  };

void main(void)
  {
    derivata2 obiect;
  }
```

Când compilați și executați programul *cons_des.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Construieste clasa baza
Construieste clasa derivata1
Construieste clasa derivata2
Distruge clasa derivata2
Distruge clasa derivata1
Distruge clasa baza
C:\>
```

1065 CUM SE TRANSMIT PARAMETRII CONSTRUCTORILOR CLASEI DE BAZĂ

C/C++

După cum știți, de fiecare dată când creați o instanță a unei clase derivate, programul dumneavoastră va apela constructorul pentru fiecare dintre clasele din care derivează clasa respectivă, pe lângă constructorul clasei derivate înseși. În multe cazuri, clasele de bază din

care derivează clasa conțin funcții constructor care așteaptă parametri. Când clasa dumneavoastră derivată nu transmite parametri către funcțiile constructor ale claselor superioare din arborele ierarhic (adică, clasa de bază din care derivează clasa), va apărea o eroare și compilatorul nu va mai compila programul. Prin urmare, C++ vă permite și totodată se așteaptă ca dumneavoastră să transmiteți parametri către funcțiile constructor superioare obiectului. Pentru a accepta declarațiile pentru clasa de bază și alte clase superioare din arbore, puteți utiliza forma generală expandată a declarației constructorului clasei derivate, așa cum arătăm mai jos:

```
derivata constructor(lista-argumente): baza1(lista-argumente),
                                       baza2(lista-argumente),
                                       baza3(lista-argumente),
                                       .
                                       .
                                       .
                                       bazaN(lista-argumente)
{
   // corpul constructorului derivatei
}
```

În declarația precedentă, *baza1* până la *bazaN* reprezintă clasele de deasupra clasei derivate, în arborele ierarhic. Pentru a înțelege mai bine modul în care veți apela constructorii clasei de bază pentru o clasă derivată, analizați următorul program, *cls_param.cpp:*

```
#include <iostream.h>
class baza
  {
   protected:
     int i;
   public:
     baza(int x)
     {
       i=x;
       cout << "Construieste clasa baza.\n";
     }
     ~baza(void) {cout << "Distruge clasa baza.\n";}
  };
class derivata : public baza
  {
    int j;
  public:
    // derivata utilizeaza x; baza utilizeaza y.
    derivata(int x, int y): baza(y)
      {
        j=x;
        cout << "Construieste clasa derivata.\n";
      }
    ~derivata(void) {cout << "Distruge clasa derivata.\n";}
```

```
    void arata(void) {cout << i << ", " << j << endl;}
 };

void main(void)
 {
   derivata obiect(3,4);
   obiect.arata(); // Afiseaza 4, 3
 }
```

Programul *cls_param.cpp* derivează clasa *derivata* din clasa de bază *baza*. Atunci când programul își începe execuția, el creează o instanță a clasei *derivata* și transmite constructorului valorile 3 și 4. Constructorul *derivata* transmite cea de-a doua valoare (în acest caz 4) către constructorul *baza*. După ce constructorul de bază își încheie execuția, constructorul *derivata* utilizează prima valoare (în cazul nostru 3) ca parametru al său. Atunci când compilați și executați programul *cls_param.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Construieste clasa baza
Construieste clasa derivata
4, 3
Distruge clasa derivata
Distruge clasa baza
C:>
```

1066 DECLARAȚIILE DE ACCES ȘI CLASELE DERIVATE



După cum ați învățat în secțiunea 1054, atunci când o clasă derivată moștenește o clasă de bază în modul *private*, toți membrii publici și protejați ai acelei clase devin membri privați ai clasei derivate, ceea ce înseamnă că membrii sunt încapsulați în cadrul clasei și programele dumneavoastră pot să acceseze membrii prin intermediul funcțiilor de interfață ale clasei derivate. Totuși, în unele cazuri, puteți să aduceți unul sau mai mulți membrii moșteniți la specificările lor de acces originare. De exemplu, puteți să acordați unor membri *publici* ai clasei de bază un statut public în clasa derivată, chiar dacă clasa derivată a moștenit clasa de bază în modul *private*. Pentru aceasta, trebuie să utilizați o *declarație de acces* în cadrul clasei derivate. Declarațiile de acces din programele dumneavoastră vor avea următoarea formă generală:

```
clasa_de_baza::membru;
```

Pentru a înțelege mai bine cum funcționează o declarație de acces, analizați următorul fragment de cod:

```
Class baza
  {
    public:
      int j;
      int k;
  };
```

```
class derivata : private baza
   {
    public:
     baza::j;
     // mai multe declaratii
   };
```

În exemplul precedent, variabila *j* ar fi fost de obicei privată în clasa *derivata* (deoarece clasa *derivata* a utilizat cuvântul cheie *private* pentru a moșteni clasa *baza*). Totuși, instrucțiunea *baza::j*, care redeclară pe *j* ca membru *public* al clasei *derivata*, redă membrului *j* un statut de membru public, fără a afecta restul clasei. Pe de altă parte, variabila *k* va rămâne privată în cadrul clasei *derivata*.

UTILIZAREA DECLARAȚIILOR DE ACCES CU CLASELE DERIVATE

C/C++1067

În secțiunea 1066 ați învățat despre declarațiile de acces din cadrul claselor derivate. În scurtul fragment de cod din acea secțiune, ați învățat bazele utilizării declarațiilor de acces. Următorul program, *acc_decl.cpp*, utilizează câteva declarații în plus pentru a avansa conceptul de declarație de acces. În programul *acc_decl.cpp*, codul convertește trei funcții *private* din cadrul clasei *derivata* înapoi la acces public. Însă, așa cum arată ultima declărație (în cadrul unui comentariu), programul nu poate declara membrul *i* înapoi la modul *public*, deoarece el este de tip *private* în cadrul clasei *baza*. Transformarea lui în membru *public* în cadrul clasei *derivata* ar fi încălcat regulile încapsulării și prin urmare compilatorul ar fi returnat o eroare la momentul atribuirii. Programul *acc_decl.cpp* vă oferă o privire de ansamblu asupra modului de lucru al declarațiilor de acces:

```
#include <iostream.h>
class baza
   {
     int i; // de tip privat in clasa baza
    public:
     int j, k;
     void seti(int x) {i = x;}
     int redai(void) {return i;}
   };
class derivata : private baza
   {
    public:
                // Urmatoarele instructiuni se suprapun
                // mostenirii private.
     baza::j;  // face din nou public pe j
     baza::seti;  // face public pe seti()
     baza::geti;  // face public pe geti()
     // baza::i; e o instructiune ilegala, nu puteti avea acces.
     int a;
   };
```

```
void main(void)
   {
     derivata obiect;
     //obiect.i = 10; Instructiune ilegala; i e privat pentru baza.
     obiect.j = 20; // legala deoarece j e public
     //obiect.k = 30; Ilegala deoarece k e privat pentru derivata
     obiect.a = 40;
     obiect.seti(10);
     cout << obiect.redai() << ", " << obiect.j << ", " << obiect.a;
   }
```

După cum puteți vedea, programul *acc_decl.cpp* derivează clasa *derivata* cu ajutorul cuvântului cheie *private* și apoi face din nou publici mare parte din membrii clasei *derivata*. Ca regulă generală, atunci când utilizați declarații de acces într-o clasă derivată, trebuie să păstrați numărul acestor declarații de acces cât mai mic sau să reconsiderați modul în care v-ați scris codul. Păstrarea unui număr minim de declarații de acces previne confuziile (atât pentru dumneavoastră cât și pentru un alt programator care vă citește codul) și face programele dumneavoastră mai conforme cu regulile încapsulării.

1068 EVITAREA AMBIGUITĂȚILOR ÎN CLASELE DE BAZĂ

C/C++

Atunci când programele dumneavoastră derivează o clasă ce moștenește mai multe clase derivate anterior dintr-o singură clasă de bază, este posibil ca acea clasă să conțină membri care, deși unici în cadrul claselor părinți, poartă același nume în cadrul clasei derivate. În astfel de situații, ambiguitatea membrilor clasei va face compilatorul să eșueze. De exemplu, dacă scrieți un program care utilizează clasa *baza* și derivează din aceasta două clase, *derivata1* și *derivata2*, programul dumneavoastră nu are ambiguități. Însă, dacă mai târziu programul dumneavoastră derivează clasa *derivata3* din ambele clase *derivata1* și *derivata2*, fiecare obiect al clasei *derivata3* va conține de fapt câte două obiecte ale clasei *baza*, după cum arătăm în figura 1068.

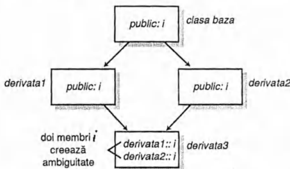


***Figura 1068** Derivarea din mai multe clase poate produce ambiguități.*

Nu numai că este inutil ca fiecare obiect al clasei *derivata3* să conțină două obiecte ale clasei *baza*, dar această ambiguitate vă derutează atât pe dumneavoastră, cât și compilatorul. De

exemplu, următorul program, *ambig_cl.cpp*, creează ambiguitate în clasa *derivata3*. Din cauza acestei ambiguități, compilatorul nu-și va încheia execuția și va returna în schimb o eroare care va forța compilatorul să se oprească:

```
// Acest program contine trei erori si nu se va compila.
#include <iostream.h>

class baza
  {
    public:
      int i;
  };

class derivata1 : public baza
  {
    public:
      int j;
  };

class derivata2 : public baza
  {
    public:
      int k;
  };

class derivata3 : public derivata1, public derivata2
  {
    public:
      int suma;
  };

void main(void)
  {
    derivata3 obiect;
    obiect.i = 10;   // Aceasta provoaca oprirea compilatorului
                     // deoarece nu stie la care i va referiti.
    obiect.j = 20;
    obiect.k = 30;
    obiect.sum = obiect.i + obiect.j + obiect.k;
    cout << obiect.i << " ";
    cout << obiect.j << " " << obiect.k << " ";
    cout << obiect.suma << endl;
  }
```

Deoarece clasa *baza* conține membrul public *i* și ambele clase *derivata1* și *derivata2* descind din clasa *baza*, clasa derivată *derivata3* conține de fapt două instanțe ale membrului public *i*. Deoarece compilatorul nu poate rezolva ambiguitatea membrului *i*, programul *ambig_cl.cpp* va returna trei erori de compilare, toate din cauza utilizării ambigue a membrului *i* în cadrul variabilei de tipul *derivata3*, *obiect*. După cum veți învăța în secțiunea 1069, programele dumneavoastră pot utiliza clase de bază virtuale pentru a preveni erori de ambiguitate în cadrul claselor derivate.

1069 CLASELE DE BAZĂ VIRTUALE



După cum ați învățat în secțiunea 1068, atunci când programele dumneavoastră derivează două sau mai multe obiecte dintr-o clasă de bază comună, pot apărea erori de ambiguitate. Pentru a preveni ambiguitatea, trebuie să preveniți apariția mai multor copii ale clasei de bază într-un obiect dat. Pentru a preveni ca programele dumneavoastră să moștenească mai multe copii ale unei clase de bază date, puteți declara acea clasă de bază utilizând cuvântul cheie *virtual*, așa cum arătăm în următorul program, *virclas.cpp*:

```
#include <iostream.h>

class baza
  {
    public:
      int i;
  };

class derivata1 : virtual public baza
  {
    public:
      int j;
  };

class derivata2 : virtual public baza
  {
    public:
      int k;
  };

class derivata3 : public derivata1, public derivata2
  {
    public:
      int suma;
  };

void main(void)
  {
    derivata3 obiect;

    obiect.i = 10; // acum i nu mai este ambiguu
    obiect.j = 20;
    obiect.k = 30;
    obiect.suma = obiect.i + obiect.j + obiect.k;
    cout << obiect.i << " ";
    cout << obiect.j << " " << obiect.k << " ";
    cout << obiect.suma << endl;
  }
```

În cazul programului *virclas.cpp*, faptul că fiecare din cele două clase derivate intermediare moștenesc clasa de bază în mod *virtual* permite clasei derivate din final, *derivata3*, să moștenească ambele clase fără să mai apară probleme de ambiguitate. După cum ați văzut în secțiunea 1068, dacă nu ar fi fost cuvântul cheie *virtual*, clasa *derivata3* ar fi avut de fapt doi

membri *i*. Ca regulă, dacă programele dumneavoastră derivează clase din mai multe clase ce împart o singură clasă părinte, trebuie să utilizați cuvântul cheie *virtual* în definiția clasei pentru a preveni ambiguitatea.

CLASELE FRIEND MUTUALE

C/C++1070

După cum ați învățat, C++ vă permite să specificați alte clase de tip *friend* ale unei clase, ceea ce permite funcțiilor din clasa *friend* să acceseze metodele private ale primei clase. Examinând programele în C++, veți întâlni cazuri în care două clase sunt *friend* una pentru cealaltă. Cu alte cuvinte, funcțiile unei clase pot accesa datele private ale celeilalte clase, iar funcțiile din această clasă pot de asemenea să acceseze datele private ale primei clase. Următorul program, *mutual.cpp*, ilustrează două clase care sunt *friend* una pentru cealaltă:

```
#include <iostream.h>
#include <string.h>

class Stan
  {
   public:
     Stan(char *msj) { strcpy(mesaj, msj); };
     void arata_mesaj(void) { cout << mesaj << endl; };
     friend class Bran;
     void arata_bran(class Bran bran);
   private:
     char mesaj[256];
  };

class Bran
  {
   public:
     Bran(char *msj) { strcpy(mesaj, msj); };
     void arata_mesaj(void) { cout << mesaj << endl; };
     friend class Stan;
     void arata_stan(class Stan stan);
   private:
     char mesaj[256];
  };

void Stan::arata_bran(class Bran bran) { cout << bran.mesaj
  << endl; };

void Bran::arata_stan(class Stan stan) { cout << stan.mesaj
  << endl; };

void main(void)
  {
    class Bran bran("Hi, hi, hi...");
    class Stan stan("Ho, ho, ho...");
    bran.arata_mesaj();
    bran.arata_stan(stan);
```

```
    stan.arata_mesaj();
    stan.arata_bran(bran);
  }
```

Deoarece programul *mutual.cpp* a utilizat calificativul *friend* la declararea ambelor clase, clasele *friend* mutuale *Stan* și *Bran* pot accesa fiecare datele private ale celeilalte. Atunci când compilați și executați programul *mutual.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Hi, hi, hi...
Ho, ho, ho...
Ho, ho, ho...
Hi, hi, hi...
C:\>
```

1071 CUM POATE O CLASĂ DERIVATĂ SĂ DEVINĂ O CLASĂ DE BAZĂ

C/C++

După cum ați învățat, C++ vă permite să creați o ierarhie de moșteniri, care permite unei clase să moștenească toate caracteristicile unei clase de bază care la rândul ei poate avea caracteristici moștenite de la o a treia clasă de bază. Următorul program, *treiniv.cpp*, derivează trei niveluri de clase. După cum veți vedea, fiecare clasa moștenește succesiv caracteristicile fiecărei clase care a fost înaintea ei:

```
#include <iostream.h>

class Baza
  {
   public:
    void arata_baza(void) { cout << "Mesajul clasei Baza\n"; };
  };

class Nivel1 : public Baza
  {
   public:
    void arata_nivel1(void)
      {
        arata_baza();
        cout << "Mesajul clasei Nivel 1\n";
      };
  };

class Nivel2 : public Nivel1
  {
   public:
    void arata_nivel2(void)
      {
        arata_nivel1();
        cout << "Mesajul clasei Nivel 2\n";
      };
```

```
  };
class Nivel3 : public Nivel2
  {
    public:
      void arata_nivel3(void)
        {
          arata_nivel2();
          cout << "Mesajul clasei Nivel 3\n";
        };
  };
void main(void)
  {
    Nivel3 datele_mele;
    datele_mele.arata_nivel3();
  }
```

După cum puteți vedea din codul programului *treiniv.cpp*, clasa *Nivel3* derivează din trei clase precedente. Pentru a înțelege modul în care clasa *Nivel3* derivează din toate cele trei clase definite anterior, considerați în ordine fiecare derivare. Clasa *Nivel3* derivează direct din clasa *Nivel2*. La rândul ei, clasa *Nivel2* derivează direct din clasa *Nivel1*. În fine, clasa *Nivel1* derivează direct din clasa *Baza*. Atunci când compilați și executați programul *treiniv.cpp*, ecranul dumneavoastră va afișa următoarele :

```
Mesajul clasei Baza
Mesajul clasei Nivel 1
Mesajul clasei Nivel 2
Mesajul clasei Nivel 3
C:\>
```

UTILIZAREA MEMBRILOR DE TIPUL PROTECTED ÎN CLASELE DERIVATE

C/C++1072

De fiecare dată când creați o clasă, trebuie să presupuneți că acea clasă va deveni o clasă de bază pentru alte derivări de clase. Prin urmare, trebuie să vă folosiți de membrii protejați pentru a limita accesul programului la membrii clasei, atât pentru instanțele clasei curente cât și pentru instanțele unei viitoare clase derivate. După cum ați învățat, membrii portejați permit clasei derivate să acceseze membrii unei clase de bază, ca și cum aceștia ar fi fost publici, dar nu permite derivări dincolo de prima accesare a membrilor clasei de bază ca membri publici. Următorul program, *protderi.cpp*, ilustrează modul de utilizare al datelor protejate într-o clasă derivată, care la rândul ei a devenit o clasă de bază:

```
#include <iostream.h>
#include <string.h>

class Baza
  {
    public:
      Baza(char *sir) { sircpy(mesaj, sir); };
      void arata_baza(void) { cout << mesaj << endl; };
```

```
    protected:
      char mesaj[256];
  };

class Nivel1 : public Baza
  {
    public:
      Nivel1(char *sir, char *baza) : Baza(baza) { sircpy(mesaj,
        sir);};
      void arata_nivel1(void) { cout << mesaj << endl; } ;
    protected:
      char mesaj[256];
  };

class De_jos : public Nivel1
  {
    public:
      De_jos(char *sir, char *nivel1, char *baza) : Nivel1(nivel1,
        baza)
          { sircpy(mesaj, sir); };
      void arata_de_jos(void)
        {
          arata_baza();
          arata_nivel1();
          cout << mesaj << endl;
        };
    protected:
      char mesaj[256];
  };

void main(void)
  {
    De_jos jos("Mesajul clasei De_jos", "Mesajul clasei Nivel1",
      "Mesajul clasei Baza");
    jos.arata_de_jos();
  }
```

După cum puteți vedea din cod, fiecare calsă definește membrul mesaj ca *protected* - ceea ce permite claselor derivate să derive acest membru, dar previne modificarea membrului mesaj în mod direct către codul din afara claselor. Deoarece fiecare clasă derivată definește membrul mesaj ca *protected*, clasele derivate pot accesa în mod direct datele dacă o doresc. În programul *protderi.cpp* însă, clasele derivate încă mai utilizează funcții de interfață. Ca regulă, programele dumneavoastră trebuie să întărească principiile încapsulării, chiar atunci când lucrează cu membri protejați dintr-o clasă derivată. Atunci când compilați și executați programul *protderi.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Mesajul clasei Baza
Mesajul clasei Nivel1
Mesajul clasei De_jos
C:\>
```

DEFINIREA DATELOR STATICE ALE UNEI CLASE

C/C++1073

După cum ați învățat, atunci când declarați membrii unei clase, C++ vă permite să precedați o definiție cu calificativul *static*. De exemplu, următoarea definiție a unei clase, utilizează membri statici și nestatici:

```
class OClasa
  {
   public:
     static int contor;
     OClasa(int valoare)
       {
         contor++;
         datele_mele=valoare;
       }
     ~OClasa(void) { contor--; };
     int datele_mele;
  };
```

În mod normal, fiecare instanță a unui obiect primește propriile sale date membre. Însă, dacă precedați o definiție a unui membru cu cuvântul cheie *static*, toate instanțele obiectului vor partaja acel membru. Dacă una dintre instanțe modifică datele, celelalte instanțe vor recunoaște imediat noua valoare a membruui. O definire a unui membru *static* nu alocă memorie pentru acel membru. În schimb, trebuie să declarați variabila statică în afara clasei, ca mai jos:

```
int OClasa::contor;
```

Următorul program, *static.cpp*, utilizează cuvântul cheie *static* pentru a partaja variabila membru *contor* (care păstrează numărul de instanțe):

```
#include <iostream.h>

class OClasa
  {
   public:
     static int contor;
     OClasa(int valoare)
       {
         contor++;
         datele_mele = valoare;
       };
     ~OClasa(void) { contor--; };
     int datele_mele;
  };

int OClasa::contor;

void main(void)
  {
```

```
    OClasa Unu(1);
    cout << "Unu: " << Unu.datele_mele << ' ' << Unu.contor
      << endl ;
    // Declara alta instanta
    OClasa Doi(2);
    cout << "Doi: " << Doi.datele_mele << ' ' << Doi.contor
      << endl ;
    // Declara alta instanta
    OClasa Trei(3);
    cout << "Trei: " << Trei.datele_mele << ' ' << Trei.contor
      << endl ;
  }
```

De fiecare dată când programul creează o nouă instanță a obiectului, constructorul clasei incrementează variabila statică *contor*. După crearea a trei instanțe, *contor* va avea valoarea 3. După cum puteți vedea, toate cele trei instanțe partajează membrul *static contor*. Atunci când compilați și executați programul *static.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Unu: 1 1
Doi: 2 2
Trei: 3 3
C:\>
```

1074 INIȚIALIZAREA UNEI DATE MEMBRE DE TIP STATIC

C/C++

În secțiunea 1073 ați învățat că C++ vă permite să declarați membri *statici* care sunt accesibili tuturor instanțelor unei clase. Atunci când utilizați date membre *statice*, trebuie să determinați cea mai bună metodă de a inițializa membrii. O modalitate este de a permite primei instanțe să transmită valoarea dorită către funcția constructor. Pentru aceasta, supraîncărcați funcția constructor pentru a accepta unul sau doi parametri. Dacă inițializatorul obiectului transmite doi parametri, funcția constructor atribuie cel de-al doilea parametru variabilei statice. Următorul program, *stat_ini.cpp*, supraîncarcă funcția constructor în acest mod pentru a inițializa membrul static la 999:

```
#include <iostream.h>

class OClasa
  {
    public:
      static int contor;
      OClasa(int valoare)
        {
          contor++;
          datele_mele = valoare;
        };
      OClasa(int valoare, int static_valoare)
        {
```

```
        contor = static_valoare;
        datele_mele = valoare;
      };
    ~OClasa(void) { contor--; };
    int datele_mele;
 };

int OClasa::contor;

void main(void)
 {
   OClasa Unu(1, 999);
   cout << "Unu: " << Unu.datele_mele << ' ' << Unu.contor
     << endl ;
   // Declara alta instanta
   OClasa Doi(2);
   cout << "Doi: " << Doi.datele_mele << ' ' << Doi.contor
     << endl ;
   // Declara alta instanta
   OClasa Trei(3);
   cout << "Trei: " << Trei.datele_mele << ' ' << Trei.contor
     << endl ;
 }
```

După cum veți învăța în secțiunea 1075, programele dumneavoastră pot de asemenea să acceseze în mod direct un membru *public* de tip *static*, pentru a atribui sau referenția valoarea membrului. Atunci când compilați și executați programul *stat_ini.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Unu: 1 999
Doi: 2 1000
Trei: 3 1001
C:\>
```

ACCESUL DIRECT LA O DATĂ MEMBRĂ DE TIP STATIC

C/C++ 1075

În secțiunea 1074 ați supraîncărcat o funcție constructor pentru a ajuta programul să inițializeze o dată membră de tip *static*. Atunci când un membru *static* este *public*, programele dumneavoastră pot accesa în mod direct valoarea membrului. Prin urmare, programul *stat_ini.cpp* ar fi putut inițializa membrul folosind două tehnici diferite. Mai întâi, programul dumneavoastră trebuie să fi atribuit valoarea atunci când a declarat membrul în afara clasei, ca mai jos:

```
int OClasa::contor = 999;
```

Apoi, în cadrul funcției *main*, programul poate accesa în mod direct membrul static, ca mai jos:

```
void main(void)
  {
    OClasa::contor = 999;
    OClasa Unu(1);
    // alte instructiuni
  }
```

Atunci când declarați un membru *static public*, programul poate accesa direct valoarea membrului, chiar dacă nu există încă instanțe ale clasei. Pentru a proteja mai bine membrii statici, utilizați membri de tip *static private*, așa cum explicăm în secțiunea 1076.

1076 DATELE MEMBRE DE TIP STATIC PRIVATE

C/C++

După cum ați învățat, C++ vă permite să declarați membrii de tip *static* pe care toate instanțele unei clase îi pot accesa. Dacă membrul este și *public*, programul însuși poate accesa membrul, evitând instanțele clasei. Pentru a proteja mai bine membrul static, îl puteți declara ca membru de tip *privat*. Atunci când membrul static este *privat*, numai funcțiile membre ale clasei pot accesa acest membru. Următorul program, *privstat.cpp*, ilustrează modul de utilizare al unui membru de tipul *private static*:

```
#include <iostream.h>

class OClasab {
  public:
    OClasa(int valoare)
      {
        contor++;
        datele_mele = valoare;
      };
    OClasa(int valoare, int static_valoare)
      {
        contor = static_valoare;
        datele_mele = valoare;
      };
    ~OClasa(void) { contor--; };
    void arata_valori(void) { cout << datele_mele << ' '
      << contor << endl; };
  private:
    static int contor;
    int datele_mele;
};

int OClasa::contor;

void main(void)
  {
    OClasa Unu(1, 999);
    Unu.arata_valori();
    // Declara alta instanta
```

```
    OClasa Doi(2, 1000);
    Doi.arata_valori();
    // Declara alta instanta
    OClasa Trei(3);
    Trei.arata_valori();
}
```

Atunci când declarați un membru de tip *static privat*, ca în programul *privstat.cpp*, programul dumneavoastră poate utiliza un constructor pentru a inițializa membrul sau programul dumneavoastră poate atribui o valoare la inițializare, care apare în afara definiției clasei.

FUNCȚIILE MEMBRE DE TIP STATIC

C/C++1077

După cum ați învățat, C++ vă permite să utilizați date membre de tip *static*, ale căror valori pot fi partajate de toate instanțele. C++ mai permite, pe lângă utilizarea datelor membre statice și utilizarea de funcții membre *statice*.Totuși, programatorii de C++ nu utilizează prea frecvent funcții membre *statice*. În general, unica întrebuințare a funcțiior membre *statice* este pentru a manipula date membre *statice*. Spre deosebire de alte funcții membre, care pot utiliza pointerul *this* pentru a accesa datele unei instanțe, funcțiile membre *statice* nu pot accesa pointerul *this* sau datele unei instanțe. Prin urmare, singura dată când veți utiliza o funcție membră *statică* este atunci când aveți o funcție care nu manipulează datele unei instanțe. Următorul program, *fnstatic.cpp*, ilustrează modul de utilizare al unei funcții membre *statice*:

```
#include <iostream.h>
class OClasa
  {
    public:
      OClasa(int valoare) { o_valoare = valoare; };
      void arata_data(void) { cout << data << ' ' << o_valoare
         << endl; };
      static void set_data(int valoare) { data = valoare; };
    private:
      static int data;
      int some_valoare;
  };
int OClasa::data;
void main(void)
  {
    OClasa clasa_mea(1001);
    clasa_mea.set_data(5005);
    clasa_mea.arata_data();
  }
```

În programul *fnstatic.cpp*, ambii membri *data* și *set_data* sunt membri statici. Cu toate că programul *fnstatic.cpp* creează doar o singură instanță a unui obiect de tipul *Oclasa*, el poate cu ușurință crea mai multe și toate vor partaja aceeași valoare statică, *5005*.

1078 ACCESUL DIRECT LA O FUNCȚIE PUBLICĂ STATICĂ

C/C++

După cum ați învățat în secțiunea 1077, C++ vă permite să definiți funcții de tip *public static* în cadrul unei clase. Atunci când declarați astfel de funcți ca publice, ele vor fi pe deplin accesibile în cadrul programului, chiar dacă programul nu a creat încă o instanță a clasei. Pentru a accesa o funcție membră de tipul *public static*, programul dumneavoastră va folosi operatorul de rezoluție globală (::), ca mai jos:

```
nume_clasa::nume_membru(parametri);
```

Următorul program, *globstat.cpp*, ilustrează cum se accesează în mod direct o funcție membră de tipul *public static*. Observați că programul folosește funcția *mesaj*, chiar dacă nu există instanțe ale obiectului *OClasa*:

```
#include <iostream.h>

class OClass
  {
    public:
      static void mesaj(void) { cout << "Salut!\n"; } ;
  };

void main(void)
  {
    OClass::mesaj();
  }
```

1079 UTILIZAREA TIPURILOR SPECIALE CA MEMBRI DE CLASĂ

C/C++

Pentru simplitate, majoritatea exemplelor prezentate în acest capitol au utilizat membrii de clasă care erau de tip *int*, *float* sau *char*. Însă, pe măsură ce definițiile claselor dumneavoastră devin mai complexe, membrii claselor dumneavoastră vor fi pointeri, referințe, tipuri enumerate sau chiar clase imbricate. Următorul program, *noi_memb.cpp*, ilustrează modul de utilizare a unor membri de clasă mai complecși:

```
#include <iostream.h>

enum Zile { luni, marti, miercuri, joi, vineri };

class CuNoroc
  {
    public:
      int *numar_cu_noroc;
      enum Zile zi_cu_noroc;
  };

void main(void)
  {
```

```
    CuNoroc oho;
    int noroc = 1500;
    oho.zi_cu_noroc = luni;
    oho.numar_cu_noroc = &noroc;
    cout << "Numarul meu norocos este " << *(oho.numar_cu_noroc)
      << endl;
    switch (oho.zi_cu_noroc)
      {
        case luni: cout << "Ziua mea norocoasa este luni\n";
              break;
         default: cout << " Ziua mea norocoasa nu este orice zi,
          ci luni\n";
      };
  }
```

Clasa *CuNoroc* definește un singur membru public de tip pointer la *int* și un singur membru de tipul enumerare *Zile*, definit chiar înaintea definiției clasei *CuNoroc* în fișierul programului. Atunci când se execută programul, el declară o instanță a clasei *CuNoroc*, denumită *oho* și o variabilă de tip *int* denumită *noroc*. Programul atribuie membrului *zi_cu_noroc* valoarea enumerată *luni* și membrului *numar_cu_noroc* o referință la variabila *noroc*. Apoi, programul afișează rezultatul atribuirii valorilor către membrii clasei *CuNoroc*. Atunci când compilați și executați programul *noi_memb.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Numarul meu norocos este 1500
Ziua mea norocoasa este luni
C:\>
```

IMBRICAREA UNEI CLASE — C/C++ 1080

După cum ați învățat în secțiunea 1079, C++ permite membrilor claselor dumneavoastră să fie de orice tip, inclusiv alte clase. Următorul program, *imbrclas.cpp*, ilustrează modul de utilizare a unei clase *imbricate*:

```
#include <iostream.h>
class Extern
  {
   public:
    Extern(void)
     {
      cout << "Tocmai am creat o instanta a unei clase din
        exterior\n";
      date_ext= 2002;
     };
    class Intern
     {
      public:
       Intern(void)
        {
```

```
                cout << "Tocmai am creat o instanta a unei clase din
                      interior\n";
                date_int = 1001;
              };
            void arata_date(void) { cout << "Intern: " << date_int
              << endl; };
          private:
            int date_int;
        } date_interne;
      void arata_toate_date(void)
        {
          date_interne.arata_date();
          cout << "Extern: " << date_interne << endl;
        };
    private:
      int date_ext;
  };
void main(void)
  {
    Extern datele_mele;
    datele_mele.arata_toate_date();
  }
```

Atunci când compilaţi şi executaţi programul *imbrclas.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Tocmai am creat o instanta a unei clase din interior
Tocmai am creat o instanta a unei clase din exterior
C:\>
```

Ca regulă, clasele *imbricate* pot deveni foarte dificil de înţeles şi ele reduce gradul de neutilizare al programului dumneavoastră. O soluţie mai bună ar fi să implementaţi două clase distincte, cu ajutorul clasei *Intern* ca o clasă de bază de la care să derivaţi clasa *Extern*.

1081 SUBCLASELE ŞI SUPERCLASELE



Citind alte cărţi sau articole despre C++ sau Java, alt limbaj de programare orientat pe obiecte, poate că aţi întâlnit termenii de *subclasă* şi *superclasă*. Aceşti termeni se referă la moştenirile unei clase. În general, termenii de subclasă şi clasă de bază sunt echivalenţi. În mod asemănător, termenii de superclasă şi clasă derivată sunt similari. Pentru explicaţiile din C++, rămâneţi la termenii de clasă *de bază* şi clasă *derivată*.

1082 INSTRUCŢIUNI INLINE ÎN LIMBAJ DE ASAMBLARE INCLUSE ÎNTR-O METODĂ



După cum aţi învăţat, majoritatea compilatoarelor de C şi C++ permit programelor dumneavoastră să plaseze instrucţiuni *inline* scrise în limbaj de asamblare în cadrul codului programului. După cum veţi învăţa în această secţiune, programele dumneavoastră pot plasa, de asemenea, instrucţiuni *inline* în limbaj de asamblare în cadrul metodelor unei clase. Următorul program,

clasbeep.cpp, creează doi membri, *beep* și *beepbeep*, care utilizează instrucțiuni *inline* scrise în limbaj de asamblare pentru a face difuzorul încorporat să emită un sunet (*beep*):

```
#include <iostream.h>

class Beeper
  {
    public:
      void beep(void);
      void beepbeep(void);
  };

void Beeper::beep(void)
  {
    asm
      {
        mov ah,2;
        mov dl,7;
        int 0x21;
      }
  }

void Beeper::beepbeep(void)
  {
    asm
      {
        mov ah,2;
        mov dl,7;
        int 0x21;
        mov ah,2;
        mov dl,7;
        int 0x21;
      }
  }

void main(void)
  {
    Beeper zgomot;
    zgomot.beep();
    zgomot.beepbeep();
  }
```

Atunci când utilizați limbajul de asamblare *inline*, majoritatea compilatoarelor de C++ vă vor cere să declarați funcțiile membre corespunzătoare în afara clasei.

MEMBRII UNEI CLASE POT FI RECURSIVI

C/C++ 1083

După cum ați învățat în capitolul despre funcții al acestei cărți, o funcție recursivă se apelează pe ea însăși pentru a efectua o operație până când se îndeplinește o anumită condiție de final. Atunci când definiți funcții membre ale unei clase, ele pot fi și recursive.

Următorul program, *strclas.cpp* creează o clasă de șiruri de caractere *ClasaSir* ce deține două funcții recursive, *sir_invers* și *lung_sir*.

```
#include <iostream.h>
#include <string.h>

class ClasaSir {
  public:
    void sir_invers(char *sir)
      {
        if (*sir)
          {
            sir_invers(sir+1);
            cout.put(*sir);
          }
      };
    int lung_sir(char *sir)
      {
        if (*sir)
          return (1 + lung_sir(++sir));
        else
          return(0);
      };
    ClasaSir(char *sir) { strcpy(ClasaSir::sir, sir); };
    char sir[256];
};

void main(void)
  {
    ClasaSir titlu("Jamsa's C/C++ Programmer's Bible");
    titlu.sir_invers(titlu.sir);
    cout << endl << "Titlul este de " <<
      titlu.lung_sir(titlu.sir) << " octeti lungime.";
  }
```

Programul *strclas.cpp* mai întâi creează o instanță a clasei *ClasaSir* denumită *titlu*. Apoi programul inversează titlul și îl afișează, iar după aceea afișează un șir de caractere conținând lungimea titlului. Atunci când compilați și executați programul *strclass.cpp*, ecranul dumneavoastră va afișa următoarele:

```
elbiB s'remmargorP ++C/C s'asmaJ
Titlul este de 32 de octeti lungime.
C:\>
```

1084 POINTERUL THIS



De fiecare dată când programul dumneavoastră creează o instanță a unei clase, C++ creează un pointer special denumit *this*, care conține adresa instanței curente a obiectului. C++ recunoaște pointerul *this* numai atunci când un membru nestatic al instanței de obiect este în curs de execuție. La rândul lor, instanțele folosesc pointerul *this* pentru a accesa metodele

În mod normal utilizarea lui *this* este transparentă, cu alte cuvinte nu este necesară. Compilatorul atribuie pe *this* și efectuează redirectările necesare în mod automat. Programele dumneavoastră nu au nevoie de obicei să utilizeze pointerul *this*, dar ele o pot face și mulți programatori recurg la el pentru mai multă claritate. Următorul program, *aratathis.cpp*, folosește pointerul *this* pentru a afișa valorile câtorva membri ai instanței. De asemenea, programul afișează valorile fără a mai utiliza pointerul *this* pentru a arăta că compilatorul inserează automat instrucțiuni pentru a efectua redirectarea corectă:

```
#include <iostream.h>
#include <string.h>

class OClasa {
  public:
    void arata_cu_this(void)
      {
        cout << "Carte: " << this->titlu << endl;
        cout << "Autor: " << this->autor << endl;
      };

    void arata_fara_this(void)
      {
        cout << "Carte: " << titlu << endl;
        cout << "Autor: " << autor << endl;
      };

    OClasa(char *titlu, char *autor)
      {
        strcpy(OClasa::titlu, titlu);
        strcpy(OClasa::autor, autor);
      };
  private:
    char titlu[256];
    char autor[256];
};

void main(void)
  {
    OClasa carte("Jamsa's C/C++ Programmer's Bible", "Jamsa &
      Klander");
    carte.arata_cu_this();
    carte.arata_fara_this();
  }
```

Atunci când compilați și executați programul *aratathis.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Carte: Jamsa's C/C++ Programmer's Bible
Autor: Jamsa & Klander
Carte: Jamsa's C/C++ Programmer's Bible
Autor: Jamsa & Klander
C:\>
```

1085 CUM DIFERĂ POINTERUL THIS DE ALȚI POINTERI

C/C++

În secțiunea 1084 ați învățat că de fiecare dată când programul dumneavoastră apelează o metodă a unei instanțe, compilatorul preatribuie un pointer special denumit *this* care indică instanța obiect. Pointerul *this* este deosebit față de alți pointeri deoarece valoarea sa se modifică la fiecare nouă apelare a instanței și prin urmare programele dumneavoastră trebuie să utilizeze cu grijă acest pointer. Examinând programe în C++, poate că ați întâlnit instrucțiuni care returnează valoarea către care indică pointerul *this*, ca mai jos:

```
return (*this);
```

În multe cazuri, compilatorul va converti valoarea pointerului într-o referință, permițând unei metode să returneze o referință la o instanță. Trebuie să verificați cu atenție valoarea de retur a membrului pentru a determina dacă metoda returnează un pointer sau o valoare de referință.

1086 LEGARE LA COMPILARE ȘI LEGARE LA EXECUȚIE

C/C++

Citind articole și cărți despre prelucrarea apelurilor de funcții, probabil că ați întâlnit termeni ca legare la compilare (*early binding*) și legare la execuție (*late binding*). Termenii descriu momentul la care adresa fiecărei funcții pe care o va apela programul dumneavoastră este *rezolvată* (adică făcută cunoscută programului). Până la acest moment, compilatorul a rezolvat toate adresele funcțiilor membre ale claselor pe care le-ați utilizat fie în momentul compilării, fie în momentul execuției. Rezolvarea adresei la acest moment este denumită legare la compilare (uneori *statică*). C++ acceptă de asemenea legare dinamică (*dynamic binding*) prin intermediul funcțiilor *virtuale*. *Legarea la execuție*, denumită și legare dinamică se realizează la momentul execuției și oferă programelor care utilizează moșteniri multiple o mai mare flexibilitate. Câteva din secțiunile ce urmează vor analiza în detaliu funcțiile virtuale. Funcțiile virtuale permit limbajului C++ să accepte polimorfismul, care v-a fost prezentat în capitolul despre obiecte al acestei cărți.

1087 POINTERI LA CLASE

C/C++

Pe măsură ce programele dumneavoastră devin mai complexe, puteți lucra cu pointeri la obiecte. De exemplu, următorul program, *ptr_ob.cpp* creează o clasă de bază și o clasă derivată simple. Programul utilizează operatorul *new* pentru a aloca în mod dinamic instanțe ale fiecărui tip de clasă și utilizează indirectarea pointerilor pentru a apela metodele fiecărei instanțe:

```
#include <iostream.h>
class Baza
  {
    public:
      void mesaj_baza(void) { cout << "Aceasta este clasa
         baza\n"; };
  };
```

```
class Derivata: public Baza
  {
    public:
      void mesaj_derivata(void) { cout << "Aceasta este clasa
         derivata\n" ; };
  };

void main(void)
  {
    Baza *pointer_baza = new Baza;
    Derivata *pointer_derivata = new Derivata;
    pointer_baza->mesaj_baza();
    pointer_derivata->derivata_mesaj();
  }
```

După cum puteți vedea, accesarea membrilor fiecărei clase prin intermediul unui pointer și al operatorului de membru este fundamental identică cu accesarea membrilor fiecărei clase prin intermediul unei instanțe de clasă și al operatorului *punct*. Însă, precum veți învăța în următoarele secțiuni, dacă folosiți un pointer pentru a accesa clasa de bază, mai târziu veți folosi același pointer pentru a accesa instanțe ale clasei derivate. Ca o alternativă, programele dumneavoastră pot folosi un pointer către o clasă derivată pentru a indica o clasă de bază. Secțiunea 1088 explică în detaliu cum se utilizează un singur pointer cu clase diferite.

FOLOSIREA ACELUIAȘI POINTER CU CLASE DIFERITE

C/C++1088

În secțiunea 1087 ați creat în mod dinamic instanțe ale claselor *Baza* și *Derivata*. Pentru aceasta, programul *ptr_ob.cpp* a utilizat două variabile pointer deosebite, una declarată ca pointer către tipul *Baza*, iar alta declarată ca pointer către tipul *Derivata*. Din fericire, atunci când programele dumneavoastră utilizează moștenirea, limbajul C++ vă permite să folosiți un pointer către clasa de bază pentru a indica clasa derivată. Însă, atunci când utilizați un pointer către clasa de bază, puteți doar să accesați membri ai clasei de bază originare, nu veți putea accesa membrii clasei derivate. Următorul program, *ptrbaza.cpp* face ca pointerul către clasa de bază să indice către clasa derivată. Apoi, el va folosi pointerul pentru a accesa membrul *mesaj_baza* al clasei de bază:

```
#include <iostream.h>

class Baza
  {
    public:
      void mesaj_baza(void) { cout << "Aceasta este clasa
         Baza\n"; };
  };

class Derivata: public Baza
  {
    public:
      void mesaj_derivata(void) { cout << "Aceasta este clasa
```

```
        Derivata\n" ; };
  };
void main(void)
  {
    Baza *pointer_baza = new Baza;
    pointer_baza->mesaj_baza();
    pointer_baza = new Derivata;
    pointer_baza->mesaj_baza();
  }
```

După cum veți învăța în secțiunea 1089, atunci când o clasă derivată și o clasă de bază au aceleași nume ale membrilor și folosiți un pointer la clasa de bază pentru a indica o clasă derivată, nu întoteauna veți obține rezultatele așteptate.

1089 CONFLICTELE DE NUME ÎNTRE CLASA DE BAZĂ ȘI CELE DERIVATE

C/C++

După cum ați învățat în secțiunea 1088, limbajul C++ vă permite să folosiți un pointer declarat ca pointer către clasa de bază pentru a indica clasa derivată. Următorul program, *numebaza.cpp* utilizează un pointer la o clasă de bază pentru a indica o clasă derivată. Clasa de bază și cea derivată dețin fiecare numele de membru *arata_mesaj*:

```
#include <iostream.h>
class Baza
  {
    public:
      void arata_mesaj(void) { cout << "Aceasta este clasa
        Baza\n"; };
  };
class Derivata: public Baza
  {
    public:
      void arata_mesaj(void) { cout << "Aceasta este clasa
        Derivata\n" ; };
  };
void main(void)
  {
    Baza *pointer_baza = new Baza;
    pointer_baza->arata_mesaj();
    pointer_baza = new Derivata;
    pointer_baza->arata_mesaj();
  }
```

Atunci când compilați și executați programul *numebaza.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este clasa Baza
Aceasta este clasa Baza
C:\>
```

Implicit, atunci când clasa de bază și cea derivată utilizează aceleași nume de funcții și utilizați un pointer la clasa de bază, compilatorul de C++ va rezolva problema indicând către funcția clasei de bază. Totuși, compilatorul poate să apeleze un membru al clasei derivate. După cum veți învăța în secțiunea 1090, pentru aceasta trebuie să folosiți funcții virtuale.

FUNCȚIILE VIRTUALE

C/C++ 1090

După cum ați învățat, atunci când o clasă moștenește metodele altei clase, se poate întâmpla ca numele membrilor claselor să fie în conflict. Dacă utilizați un pointer către o clasă de bază pentru a accesa o clasă derivată și apelați unul dintre membrii cu același nume cu al unui membru al clasei de bază, se va executa membrul clasei de bază. Însă dacă doriți ca C++ să apeleze membrul clasei derivate, trebuie să definiți o *funcție virtuală* pentru acel membru al clasei de bază. Utilizarea funcțiilor virtuale nu este cu mult diferită de operațiile pe care le-ați efectuat până acum. Pentru a crea o funcție virtuală, pur și simplu precedați numele funcției cu cuvântul cheie *virtual*. Tipul de retur al funcției și lista parametrilor trebuie să fie identică pentru fiecare dintre funcțiile virtuale. Următorul program, *virt_unu.cpp* definește funcția *arata_mesaj* ca funcție virtuală în cadrul claselor *baza* și *derivata*:

```
#include <iostream.h>
class Baza
  {
   public:
    virtual void arata_mesaj(void) { cout <<"Aceasta este
       clasa Baza\n"; };
  };
class Derivata: public Baza
  {
   public:
    virtual void arata_mesaj(void) { cout <<"Aceasta este
       clasa Derivata\n" ; };
  };
void main(void)
  {
   Baza *pointer_baza = new Baza;
   pointer_baza->arata_mesaj();
   pointer_baza = new Derivata;
   pointer_baza->arata_mesaj();
  }
```

Atunci când compilați și executați programul *vrt_unu.cpp* ecranul dumneavoastră va afișa următorul rezultat:

```
Aceasta este clasa Baza
Aceasta este clasa Derivata
C:\>
```

După cum puteţi vedea, deoarece ambele clase folosesc funcţii virtuale, pointerul poate apela în mod corect metodele claselor *Baza* şi *Derivata*.

1091 MOŞTENIREA ATRIBUTULUI VIRTUAL



Atunci când clasele din programul dumneavoastră moştenesc o funcţie virtuală, funcţia moştenită menţine atributul virtual al funcţiei de bază. Cu alte cuvinte, o funcţie virtuală va rămâne virtuală indiferent de câte ori este moştenită în programul dumneavoastră. De exemplu, atunci când o clasă derivată moşteneşte o funcţie virtuală şi apoi programul dumneavoastră foloseşte acea clasă derivată ca o clasă de bază, funcţia de două ori derivată poate încă să se suprapună funcţiei virtuale. Pentru a înţelege mai bine cum poate membrul din cadrul clasei derivate să se suprapună funcţiei virtuale, analizaţi următorul program, *vfunc_su.cpp*:

```
#include <iostream.h>

class baza
  {
   public:
    virtual void vfunc(void) { cout << "Aceasta este functia
      vfunc() a clasei baza." << endl; }
  };

class derivata1 : public baza
  {
   public:
    void vfunc(void) { cout << "Aceasta este functia vfunc() a
      clasei derivata1." << endl; }
  };

class derivata2 : public derivata1
  {
   public:
    void vfunc(void) { cout << "Aceasta este functia vfunc() a
      clasei derivata2." << endl; }
  };

void main(void)
  {
   baza *p, b;
   derivata1 d1;
   derivata2 d2;
   p = &b; // Indica clasa de baza
   p->vfunc();
   p = &d1; // Indica clasa derivata1
   p->vfunc();
   p = &d2; // Indica clasa derivata2
   p->vfunc();
  }
```

În ambele cazuri, (*derivata1* și *derivata2*), definiția din interiorul clasei pentru *vfunc* se suprapune peste definiția virtuală din cadrul clasei *baza*. Prin urmare, atunci când programul *vfunc_su.cpp* modifică pointerul pentru a indica clasele derivate din secvență, accesarea membrului *vfunc* determină execuția funcțiilor locale *vfunc* pe care le definește fiecare clasă. Atunci când compilați și executați programul *vfunc_su.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Aceasta este functia vfunc() a clasei baza
Aceasta este functia vfunc() a clasei derivata1
Aceasta este functia vfunc() a clasei derivata2
C:\>
```

FUNCȚIILE VIRTUALE SUNT IERARHICE

C/C++1092

După cum ați învățat în secțiunile precedente, moștenirea este ierarhizată. Din această cauză, funcțiile virtuale trebuie de asemenea să fie ierarhizate. Pentru că funcțiile virtuale sunt ierarhizate, dacă o clasă derivată nu suprapune funcția virtuală, compilatorul va utiliza cea mai apropiată versiune superioară din arborele ierarhic. În următorul program, de exemplu, *derivata2* este derivată din *derivata1*, care la rândul ei este derivată din calsa *baza*. Totuși, clasa *derivata2* nu suprapune funcția *vfunc*, astfel încât compilatorul va folosi în schimb funcția *vfunc* suprapusă în cadrul definiției clasei *derivata1*. Atunci când lucrați cu funcții virtuale, puteți să testați ierarhia moștenirilor utilizând un program ca *virt_ier.cpp*, prezentat mai jos:

```
#include <iostream.h>
class baza
  {
   public:
    virtual void vfunc(void) { cout << "Aceasta este functia
      vfunc() a clasei baza." << endl; }
  };
class derivata1 : public baza
  {
   public:
    void vfunc(void) { cout << "Aceasta este functia vfunc() a
      clasei derivata1." << endl; }
  };
class derivata2 : public derivata1 { };
void main(void)
  {
   baza *p, b;
   derivata1 d1;
   derivata2 d2;
   p = &b; // Indica clasa baza
   p->vfunc();
   p = &d1; // Indica clasa derivata1
   p->vfunc();
```

```
    p = &d2; // Indica clasa derivata2
    p->vfunc(); // Utilizeaza totusi, functia vfunc a clasei
      derivata1
  }
```

În cazul programului *virt_ier.cpp*, modificarea pointerului către clasa *derivata2* nu afectează funcțiile apelate de operatorul de membru, deoarece clasa *derivata2* nu își definește propria implementare a membrului *vfunc*. În schimb, ultimul acces apelează membrul *vfunc* pentru clasa *derivata1*. Atunci când compilați și executați programul *virt_ier.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Aceasta este functia vfunc() a clasei baza
Aceasta este functia vfunc() a clasei derivata1
Aceasta este functia vfunc() a clasei derivata1
C:\>
```

1093 IMPLEMENTAREA POLIMORFISMULUI



După cum ați citit în capitolul despre obiecte, polimorfismul este capacitatea unui singur obiect de a lua diferite forme. Limbajul C++ acceptă polimorfismul prin intermediul funcțiilor virtuale. Cu ajutorul funcțiilor virtuale, același pointer poate indica diferite clase pentru a realiza diferite operații. Următorul program, *polimorf.cpp*, creează o clasă de bază și două clase derivate. Apoi, programul *polimorf.cpp* utilizează pointerul *poli* pentru a apela diverse metode:

```
#include <iostream.h>
#include <stdlib.h>

class Baza
  {
   public:
     virtual int add(int a, int b) { return(a + b); };
     virtual int scad(int a, int b) { return(a - b); };
     virtual int inmult(int a, int b) { return(a * b); };
  };

class TestMatem : public Baza
  {
     virtual int inmult(int a, int b)
       {
         cout << a * b << endl;
         return(a * b);
       };
  };

class ScadPoz : public Baza
  {
    virtual int scad(int a, int b) { return(abs(a - b)); };
  };

void main(void)
```

```
{
  Baza *poli = new TestMatem;
  cout << poli->add(562, 531) << ' ' << poli->scad(1500, 407)
    << endl;
  poli->inmult(1093, 1);
  poli = new ScadPoz;
  cout << poli->add(892, 201) << ' ' << poli->scad(0, 1093)
    << endl;
  cout << poli->inmult(1, 1093);
}
```

Programul *polimorf.cpp* vă arată cu claritate cum funcțiile virtuale și polimorfismul extind puterea oferită de moștenire pentru a permite programelor dumneavoastră să obțină rezultate interesante. Notați că, în funcție de instanța indicată de pointerul *poli*, operațiile pe care el le efectuează pot fi diferite. Atunci când compilați și executați programul *polimorf.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
1093 1093
1093
1093 1093
1093
C:\>
```

FUNCȚIILE VIRTUALE PURE

C/C++1094

Ați învățat că atunci când derivați o clasă din alta, C++ vă permite să utilizați funcții virtuale pentru a controla care dintre funcțiile claselor va fi apelată de program atunci când utilizați un pointer al clasei de bază pentru a indica o clasă derivată. Atunci când ați citit mai multe despre funcțiile virtuale, probabil că ați întâlnit termenul de *funcție virtuală pură*. O funcție virtuală pură este similară unui prototip pe care îl declarați în clasa de bază și care impune clasei derivate să dețină o implementare a sa. În cadrul clasei de bază, o funcție virtuală pură apare ca mai jos:

```
virtual tip nume_functie(parametri) = 0;
```

Simbolul egal (=) și valoarea 0 care urmează prototipului indică faptul că funcția este o funcție virtuală pură pentru care programul trebuie să furnizeze o implementare. Următorul program, *virtpura.cpp*, ilustrează o funcție virtuală pură:

```
#include <iostream.h>
#include <string.h>
class Baza
 {
   public:
    virtual void arata_mesaj(void) { cout << "Mesajul clasei
      Baza" << endl; };
    virtual void arata_invers(void) = 0;
 };
class Derivata : public Baza
```

```
   {
     public:
       virtual void arata_mesaj(void) { cout <<"Mesajul clasei
         Derivata" << endl; };
       virtual void arata_invers(void)
         { cout << strrev("Mesajul clasei Derivata") << endl; };
   };
 void main(void)
   {
     Baza *poli = new Derivata;
     poli->arata_mesaj();
     poli->arata_invers();
   }
```

Este important să observați că o funcție virtuală pură *necesită* ca fiecare clasă derivată să definească propria sa implementare a funcției, pe când o funcție virtuală permite claselor derivate să utilizeze funcția prevăzută în clasa de bază. În programul *virtpura.cpp* de exemplu, dacă pointerul *poli* ar fi indicat *Baza*, programul dumneavoastră nu ar fi putut să acceseze funcția membru *arata_invers*. Pointerul *poli* poate accesa această funcție doar în clasa *derivata*.

1095 CLASELE ABSTRACTE



În secțiunea 1094 ați învățat că o funcție virtuală pură este un prototip de funcție pentru care clasa de bază impune claselor derivate să conțină o implementare. Atunci când o clasă conține doar funcții virtuale pure, limbajul C++ se referă la acea clasă ca la o *clasă abstractă*. În general, o clasă abstractă pune la dispoziție un șablon pe baza căruia programele dumneavoastră pot deriva ulterior alte clase. Limbajul C++ nu vă va permite să creați o variabilă al cărei tip să fie clasă abstractă. Dacă încercați să faceți aceasta, compilatorul va genera o eroare de sintaxă.

Clasele abstracte sunt cunoscute și sub numele de *fabrici de clase*, deoarece programele dumneavoastră le utilizează ca locații centrale din care derivează (sau fabrică, pentru a păstra analogia) alte clase pentru nevoile programelor.

1096 UTILIZAREA FUNCȚIILOR VIRTUALE



După cum ați învățat, unul dintre aspectele centrale ale programării orientate spre obiecte este „o interfață, mai multe metode". Cu alte cuvinte, puteți crea clase de bază în C++ pe care programele dumneavoastră le vor utiliza pentru a defini natura interfeței unei clase generale. Fiecare clasă pe care o veți deriva ulterior dintr-o clasă de bază implementează operații specifice, corespunzătoare tipului de date pe care tipul derivat îl utilizează.

Una dintre cele mai puternice modalități prin care programele dumneavoastră pot atinge scopul „o interfață, mai multe metode" este de a utiliza funcții virtuale, clase abstracte și polimorfismul la momentul execuției. Programele care utilizează toate aceste facilități vă permit să creați o ierarhie de clase, de la cea generală la cea complexă (de la bază la derivată). Veți crea toate facilitățile uzuale precum și interfața într-o clasă de bază. În cazurile în care puteți implementa anumite acțiuni numai în cadrul unei clase derivate, clasa de bază

trebuie să definească o funcție virtuală pentru a crea interfața pe care clasele derivate o vor utiliza. Pentru a înțelege mai bine principiul „o interfață, mai multe metode", analizați următorul program, *vfunc_ex.cpp*, care aplică acest concept unor clase simple:

```
#include <iostream.h>

class conversie
  {
    protected:
      double val1;
      double val2;
    public:
      conversie(double i) { val1 = i; }
      double redaconv(void) {return val2;}
      double redainit(void) {return val1;}
      virtual void calcul(void) = 0;
  };
// litri in galoane
class l_in_g : public conversie
  {
    public:
      l_in_g(double i) : conversie(i) { }
      void calcul(void) { val2 = val1 / 3.7854; }
  };
// Fahrenheit in Celsius
class f_in_c : public conversie
  {
    public:
      f_in_c(double i) : conversie(i) { }
      void calcul(void) { val2 = (val1 - 32) / 1.8; }
  };
void main(void)
  {
    conversie *p; // pointer la clasa de baza
    l_in_g lgob(4);
    f_in_c fcob(70);
    p = &lgob; // converteste litri in galoane
    cout << p->redainit() << " litri inseamna ";
    p->calcul();
    cout << p->redaconv() << " galoane." << endl;
    p = &fcob; // converteste fahrenheit in celsius
    cout << p->redainit() << " grade Fahrenheit inseamna ";
    p->calcul();
    cout << p->redaconv() << " grade Celsius." << endl;
  }
```

Programul *vfunc_ex.cpp* derivează clasele *l_in_g* (litri în galoane) și *f_in_c* (Farenheit în Celsius) din clasa de bază *conversie*. Ambele clase derivate inițializează membrul *i* din cadrul clasei *conversie*. Ambele clase suprascriu de asemenea funcția virtuală pură *calcul*, însă ele utilizează funcțiile *redainit* și *redaconv* definite în clasa de bază. De asemenea, ambele clase derivate utilizează membrii *val1* și *val2* ai clasei de bază. De fapt, singura funcție care diferă între cele două clase derivate este membrul *calcul*, care utilizează calcule diferite pentru fiecare clasă derivată în parte. Când compilați și executați programul *.cpp*, ecranul dumneavoastră va afișa următoarele:

```
4 litri inseamna 1.05669 galoane
70 grade Farenheit inseamna 21.1111 grade Celsius
C:\>
```

1097 MAI MULTE DESPRE LEGAREA LA COMPILARE ȘI LEGAREA LA EXECUȚIE

C/C++

Atunci când discutați cu alți programatori despre programarea în C++, veți auzi probabil frecvent expresiile *legare la compilare* și *legare la execuție* (*early binding* și *late binding*). Termenul de legare la compilare (*early binding*) se referă la evenimentele ce au loc la momentul compilării. Cu alte cuvinte, compilatorul cunoaște toate informațiile care îi trebuie pentru a compila programul în timpul compilării. Exemple tipice de legare la compilare sunt: apelurile de funcții normale (cum ar fi apelurile funcțiilor de bibliotecă standard), apelurile de funcții supraîncărcate și operatorii supraîncărcați.

Pe de altă parte, termenul de *legare la execuție (late binding)* se referă la evenimente pe care programul dumneavoastră nu le rezolvă decât la momentul execuției. Pentru ca programele dumneavoastră să poată utiliza *legarea la execuție*, clasele din aceste programe trebuie să declare funcții virtuale. După cum știți, atunci când accesați o funcție virtuală prin intermediul unui pointer către bază, programul stabilește care dintre funcțiile virtuale o va apela, în funcție de obiectul curent la care indică pointerul de bază. Deoarece compilatorul nu are cum să știe ce obiect este referențiat de pointer în momentul compilării, programul trebuie să răspundă la referențierile pointerului în momentul executării lui.

1098 CUM ALEGEM ÎNTRE LEGAREA LA COMPILARE ȘI LEGAREA LA EXECUȚIE

C/C++

După cum ați învățat în secțiunea 1097, programele dumneavoastră vor cuprinde în general legări la compilare, legări la execuție sau pe amândouă. Atunci când veți scrie programe mai complexe, veți observa că legarea la execuție este utilizată mai des în cadrul programelor mai complexe decât în cele simple. Pe măsură ce adăugați mai multe funcții virtuale programelor dumneavoastră, va trebui să alegeți mai frecvent între legarea la compilare și legarea la execuție.

Principalul avantaj al legării la compilare este eficiența. Deoarece compilatorul poate rezolva întreaga funcție înainte de execuția programului, funcțiile legate la compilare sunt foarte eficiente. De asemenea, legarea la compilare este mai puțin expusă erorilor din timpul execuției, deoarece compilatorul poate detecta dinainte multe dintre problemele ce pot apărea. Pe de altă parte, principalul avantaj al legării la execuție este flexibilitatea. Spre deosebire de legarea la compilare, cea la execuție permite programelor dumneavoastră să răspundă la evenimente ce au loc în timpul execuției, fără a fi nevoie de mult „cod circumstanțial".

UN EXEMPLU DE PROGRAM CU LEGARE LA EXECUȚIE

C/C++1099

În secțiunile precedente ați creat clase ce utilizează și clase care nu utilizează funcții virtuale. După cum ați învățat în secțiunile 1097 și 1098, programatorii descriu programele ce utilizează funcții virtuale ca programe cu legare la execuție. Programele care nu folosesc funcții virtuale sunt programe cu legare la compilare. Următorul fragment de cod, de exemplu, vă arată cum trebuie să definiți clasa *conversie* din secțiunea 1096 cu o funcție virtuală:

```
class conversie
  {
    protected:
      double val1;
      double val2;
    public:
      conversie(double i) { val1 = i; }
      double redaconv(void) {return val2;}
      double redainit(void) {return val1;}
      virtual void calcul(void) = 0;
  };
// litri in galoane
class l_in_g : public conversie
  {
    public:
      l_in_g(double i) : conversie(i) { }
      void calcul(void){ val2 = val1 / 3.7854; }
  };
```

Altfel, programul dumneavoastră ar putea declara clasa *l_in_g* cu legare la compilare, ca mai jos:

```
class l_in_g : public conversie {
protected:
    double val1;
    double val2;
public:
    l_in_g(double i) { val1 = i; }
    double redaconv(void) {return val2;}
    double redainit(void) {return val1;}
    void calcul(void){ val2 = val1 / 3.7854; }
};
```

După cum puteți vedea, în fragmentul de cod precedent, legarea la execuție nu este utilă în mod special. Totuși, dacă ați fi adăugat și alte conversii programului, ați fi creat probabil mai multe clase repetitive, ce partajau aceeași prelucrare. Tot așa cum probabil ați utilizat moștenirea pentru a rezolva problema claselor repetitive, ar trebui să utilizați legarea la execuție pentru ca programele dumneavoastră să fie mai eficiente și mai clare pentru cel ce le citește.

1100 DEFINIREA UNUI MANIPULATOR AL UNUI FLUX DE IEȘIRE



Unele dintre secțiunile din capitolul de familiarizare cu limbajul C++ au utilizat manipulatori ai fluxurilor de ieșire cum ar fi *hex* și *endl*. După cum ați învățat anterior, puteți crea proprii dumneavoastră manipulatori ai fluxurilor de ieșire. De exemplu, să presupunem că doriți să scrieți un program ce creează un nou manipulator al fluxului de ieșire, denumit *atentie*, care produce un zgomot în difuzorul încorporat în calculator pentru a atrage atenția utilizatorului. Apoi veți putea utiliza manipulatorul *atentie* în cadrul fluxului *cout*, ca mai jos:

```
cout << atentie << "Cred ca discul dumneavoastra este defect!";
```

Următorul cod implementează *manipul.cpp*, care creează manipulatorul *atentie*:

```
#include <iostream.h>

ostream& atentie(ostream& cout) { return(cout << '\a'); };

void main(void)
 {
   cout << atentie << "Seful vine spre biroul tau...\n";
 }
```

1101 ESTE TIMPUL SĂ ARUNCĂM O PRIVIRE CĂTRE IOSTREAM.H



În capitolul care v-a familiarizat cu limbajul C++, vi s-a spus să nu studiați în fișierul antet *iostream.h*. În prezent, după ce stăpâniți clasele, supraîncărcarea, funcțiile virtuale și bazele programării orientate pe obiect, trebuie nu numai să vă aruncați o privire asupra acestui fișier, ci chiar să începeți să analizați fiecare linie. Cei care au scris compilatorul de C++ au utilizat câteva tehnici de programare foarte interesante atunci când au scris fișierul *iostream.h* – tehnici pe care le puteți utiliza în cadrul programelor dumneavoastră. În primul rând, tipăriți la imprimantă o copie a fișierului pe care o veți putea consulta pe măsură ce creați propriile dumneavoastră clase. În al doilea rând, faceți o copie a fișierului sub numele de *iostream.nts*. Apoi, citiți părți din fișier în fiecare zi și adăugați comentarii care explică prelucrările din fișier. Dacă parcurgeți două pagini pe zi, veți încheia studiul în mai puțin de o săptămână și nu numai că veți fi un expert în operațiile de I/O cu C++, dar veți învăța multe despre operațiile de intrare și ieșire ale claselor mai complexe.

1102 UTILIZAREA OPERATORULUI SIZEOF CU O CLASĂ



După cum știți, operatorul *sizeof* returnează numărul de octeți necesari pentru a memora un obiect. Atunci când programele dumneavoastră lucrează cu obiecte, puteți să cunoașteți dimensiunea unui obiect. De exemplu, să presupunem că citiți un fișier de obiecte. Puteți utiliza operatorul *sizeof* pentru a determina dimensiunea fiecărui obiect din cadrul fișierului. Operatorul *sizeof* returnează doar dimensiunea datelor membre ale clasei. Următorul program, *sizeof.cpp* utilizează operatorul *sizeof* pentru a determina dimensiunile a două clase. Prima clasă este o clasă de bază, iar cea de-a doua este o clasă derivată:

```
#include <iostream.h>
#include <string.h>

class Baza {
  public:
    Baza(char *mesaj) { strcpy(Baza::mesaj, mesaj); };
    void arata_baza(void) { cout << mesaj << endl; };
  private:
    char mesaj[256];
};

class Derivata: public Baza {
  public:
    Derivata(char *msjd, char *msjb) : Baza(msjb) {
      strcpy(mesaj, msjd); };
    void arata_derivata(void)
      {
        cout << mesaj << endl;
        arata_baza();
      };
  private:
    char mesaj[256];
  };

void main(void)
  {
    Baza o_baza("Aceasta este o baza");
    Derivata o_derivata("Mesaj din Derivata", "Mesaj din Baza");

    cout << "Dimensiunea clasei de baza e " << sizeof(o_baza) <<
              "octeti" << endl;
    cout << "Dimensiunea clasei derivate e " << sizeof(o_derivata)
           << " bytes" << endl;
  }
```

Atunci când compilați și executați programul *sizeof.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Dimensiunea clasei de baza este 256 octeti
Dimensiunea clasei derivate este 512 octeti
C:\>
```

ATRIBUTELE PRIVATE, PUBLIC ȘI PROTECTED SE POT APLICA ȘI STRUCTURILOR C/C++ 1103

Câteva din secțiunile acestui capitol au utilizat membri de clase privați, publici și protejați. Atunci când programele dumneavoastră în C++ folosesc structuri, puteți de asemenea avea membri privați, publici și protejați. Implicit, toți membrii unei structuri sunt publici. Însă, puteți utiliza etichetele *private* și *protected* pentru a identifica membrii cărora doriți să le controlați accesul. Următorul program, *privstru.cpp*, ilustrează modul de utilizare al

membrilor privați dintr-o structură. După cum veți vedea, majoritatea facilităților pe care C++ le oferă claselor sunt disponibile și structurilor:

```
#include <iostream.h>
#include <string.h>

struct CarteaMea {
   char titlu[64]; // Public by default
   void arata_carte(void)
     {
       cout << "Carte: " << titlu << " Pret: $" << pret ;
     };
   void da_pret(float amount) { pret = amount; };
   void atrib_titlu(char *nume) { strcpy(titlu, nume); };
   private:
     float pret;
};

void main(void)
   {
     CarteaMea carte;

     carte.atrib_titlu("Jamsa's C/C++ Programmer's Bible");
     carte.da_pret(49.95);
     carte.arata_carte();
   }
```

Atunci când compilați și executați programul *privstru.cpp*, el va construi obiectul *carte*, îi va atribui un titlu și un preț, iar apoi va afișa pe ecran informațiile despre titlul și prețul obiectului *carte*, ca mai jos:

```
Carte: Jamsa's C/C++ Programmer's Bible Pret: $49.95
C:\>
```

În programul *privstru.cpp*, structura *CarteaMea* specifică membrul *pret* ca privat. Prin urmare, singura modalitate de a accesa membrul este de a utiliza una dintre metodele publice ale structurii. Dacă veți considera că structurile dumneavoastră necesită acest tip de protecție a membrilor, ar trebui să le înlocuiți cu clase.

1104 *CONVERSIILE CLASELOR*



După cum știți, atunci când transmiteți o valoare de tipul *int* către o funcție care necesită o valoare *long*, C++ va promova valoarea întreagă la tipul corect. Asemănător, dacă transmiteți un parametru de tipul *float* unei funcții care necesită o valoare de tipul *double*, C++ va efectua o conversie similară. Atunci când lucrați cu clase în C++, puteți de asemenea specifica acele conversii pe care C++ trebuie să le efectueze pentru a converti valorile claselor într-un tip standard de date (cum ar fi *int* sau *long*) sau chiar într-o clasă diferită. Conversiile sunt necesare de obicei atunci când transmiteți un parametru de un anumit tip către o funcție constructor pentru o clasă de un tip diferit. De exemplu, să presupunem că avem clasa *DateCarte* care primește în mod normal trei șiruri de caractere ca parametri ai constructorului său:

```
DateCarte (char *titlu, char *autor, char *editura)
   {
      // Instructiuni
   }
```

Să presupunem însă că programul apelează periodic constructorul cu o structură de tipul *InfoCarte*, ca mai jos:

```
struct InfoCarte
   {
     char titlu[64];
     char autor[64];
     char editura[64];
     float pret;
     int pagini;
   };
```

Clasa poate crea un al doilea constructor care convertește datele în mod corespunzător. Secțiunea 1105 prezintă un program care utilizează o structură și un al doilea constructor.

CONVERTIREA DATELOR ÎNTR-UN CONSTRUCTOR

C/C++1105

După cum ați învățat, uneori veți fi nevoit să convertiți date dintr-un anumit format într-un format pe care clasa îl așteaptă. O modalitate simplă de a efectua o astfel de conversie este de a utiliza funcții constructor diferite. Următorul program, *convert.cpp*, utilizează o astfel de funcție constructor pentru a converti informațiile conținute de o structură de tipul *InfoCarte*:

```
#include <iostream.h>
#include <string.h>

struct InfoCarte {
   char titlu[64];
   char editura[64];
   char autor[64];
   float pret;
   int pagini;
};

class DateCarte {
   public:
     DateCarte(char *titlu, char *editura, char *autor);
     DateCarte(struct InfoCarte);
     void arata_carte(void)
       {
         cout << "Carte: " << titlu << " de " <<
         autor << endl << "Editura: " << editura << endl;
       };
     private:
```

```
      char titlu[64];
      char autor[64];
      char editura[64];
   };

DateCarte::DateCarte(char *titlu, char *editura, char *autor)
   {
     strcpy(DateCarte::titlu, titlu);
     strcpy(DateCarte::editura, editura);
     strcpy(DateCarte::autor, autor);
   }

DateCarte::DateCarte(InfoCarte carte)
   {
     strcpy(DateCarte::titlu, carte.titlu);
     strcpy(DateCarte::editura, carte.editura);
     strcpy(DateCarte::autor, carte.autor);
   }

void main(void)
   {
     InfoCarte carte = {"Rescued by C++", "Jamsa Press", "Jamsa",
                          29.95, 256 };
     DateCarte carte_mare("Jamsa's C/C++ Programmer's Bible", "Jamsa
                            Press", "Jamsa & Klander");
     DateCarte carte_mica(carte);
     carte_mare.arata_carte();
     carte_mica.arata_carte();
   }
```

Programul *convert.exe* își începe procesarea prin crearea unei instanțe a clasei *InfoCarte* în obiectul *carte*. Declarația apelează constructorul *InfoCarte*, care atribuie parametrii variabilelor membre ale obiectului *carte*. Apoi programul efectuează prelucrări similare cu obiectul *carte_mare*. Cea de-a treia declarație, *carte_mica(carte)*, apelează constructorul supraîncărcat al clasei *DateCarte*, care își extrage valorile necesare din cadrul obiectului *carte* și atribuie valorile noului obiect *carte_mica*. Atunci când compilați și executați programul *convert.exe*, ecranul dumneavoastră va afișa următorul rezultat:

```
Carte: Jamsa's C/C++ Programmer's Bible de Jamsa & Klander
Editura: Jamsa Press
Carte: Rescued by C++
Editura: Jamsa Press
C:\>
```

Utilizarea funcției constructor pentru a efectua conversia, așa cum am arătat în această secțiune, nu este diferită în realitate de operațiile de supraîncărcare pe care le-ați efectuat pe parcursul acestei cărți.

ATRIBUIREA UNEI CLASE ALTEI CLASE

C/C++1106

În secțiunea 1105 ați creat o funcție constructor care convertea datele dintr-o structură de tipul *InfoCarte* în câmpurile de date ale unei instanțe a clasei *DateCarte*. Examinând programele în C++, poate că ați întâlnit instrucțiuni în care programul atribuia un tip de clasă altei clase, ca mai jos:

```
class DateCarte carte_mare;
char titlu[256];

titlu = carte_mare;
```

În acest caz, programul atribuie clasa *carte_mare* unei variabile de tip șir de caractere. Pentru a accepta astfel de operații, programul trebuie să indice compilatorului conversia corectă pe care trebuie să o aplice. Pentru aceasta, trebuie să creați o funcție membră a clasei *DateCarte* care să efectueze conversia. În acest caz, funcția va atribui titlul cărții șirului de caractere. Există două reguli pe care funcțiile de conversie trebuie să le urmeze. Prima: în cadrul clasei, programul trebuie să definească funcția ca o funcție operator supraîncărcată, ca mai jos:

```
    operator char *(void);
```

A doua: codul funcției corespunzătoare trebuie să returneze o valoare de tipul convertit, care, în cazul nostru, este un pointer la un șir de caractere. Următorul program, *clasaatr.cpp* utilizează o funcție membră de conversie pentru a atribui clasa unui șir de caractere. În acest caz, funcția de conversie atribuie titlul cărții șirului de caractere:

```
#include <iostream.h>
#include <string.h>

class DateCarte {
  public:
    DateCarte(char *titlu, char *editura, char *autor);
    void arata_carte(void)
      {
        cout << "Carte: " << titlu << " de " <<
                autor << " Editura: " << editura << endl;
      };
    operator char *();

  private:
    char titlu[64];
    char autor[64];
    char editura[64];
  };

DateCarte::DateCarte(char *titlu, char *editura, char *autor)
  {
    strcpy(DateCarte::titlu, titlu);
    strcpy(DateCarte::editura, editura);
    strcpy(DateCarte::autor, autor);
  }
```

```
DateCarte::operator char *(void)
   {
     char *ptr = new char[256];
     return(strcpy(ptr, titlu));
   }
void main(void)
   {
     DateCarte carte_mare("Jamsa's C/C++ Programmer's Bible",
                    "Jamsa Press", "Jamsa & Klander");
     char *titlu;
     titlu = carte_mare;
     cout << "Titlul cartii este " << titlu << endl;
   }
```

Utilizând funcții de conversie ca în programul *clasaatr.cpp*, programele dumneavoastră pot converti după necesități, dintr-o clasă în alta. Atunci când compilați și executați porgramul *clasaatr.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Titlul cartii este Jamsa's C/C++ Programmer's Bible
C:\>
```

1107 UTILIZAREA FUNCȚIILOR FRIEND PENTRU CONVERSII

C/C++

Atunci când efectuați conversii între o clasă și alta, uneori va fi nevoie să accesați membrii privați ai altei clase. După cum ați învățat, puteți specifica o clasă sau o funcție *friend* pentru a permite altei clase accesul la datele private fără a face datele vizibile altor părți din program. Dacă o clasă trebuie să acceseze datele private ale altei clase pentru a efectua o conversie, specificați funcția de convertire ca fiind *friend* pentru clasă, ca mai jos:

```
friend void Conversie(DateCarte &noua, CarteInfo carte);
// Definitia clasei
void Conversie(DateCarte &noua, CarteInfo carte);
   {
     strcpy(noua.titlu, carte.titlu);
     strcpy(noua.editura, carte.editura);
     strcpy(noua.autor, carte.autor);
   }
```

1108 CUM DETERMINĂM DACĂ OPERATORII MĂRESC SAU SCAD LIZIBILITATEA

C/C++

După cum ați învățat, atunci când definiți o clasă, limbajul C++ vă permite să supraîncărcați unul sau mai mulți operatori. Totuși trebuie să determinați dacă supraîncărcarea va face programul mai simplu sau mai dificil de înțeles. De exemplu, următorul program, *sir_plus.cpp*, supraîncarcă operatorul plus pentru clasa *Sir*. Apoi programul utilizează atât operatorul, cât și funcția *siradg*, pentru a adăuga conținutul unui șir de caractere la un altul. Examinați programul și determinați care dintre tehnici este mai inteligibilă:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Sir {
  public:
    char *operator +(char *append_str)
      { return(strcat(buffer, append_str)); };

    Sir(char *sir)
      {
        strcpy(buffer, sir);
        lung = strlen(buffer);
    }

    void arata_sir(void) { cout << buffer; };
    void siradg(char *sursa) { strcat(buffer, sursa); };

  private:
    char buffer[256];
    int lung;
};

void main(void)
  {
    Sir titlu("Jamsa's C/C++ ");
    titlu = titlu + "Programmer's Bible\n";
    titlu.arata_sir();
    Sir carte2("Rescued by C++");
    carte2.siradg(" Third Edition ");
    carte2.arata_sir();
  }
```

Mulți programatori începători în C++ supraîncarcă de obicei mai mulți operatori decât este nevoie. Decizia pe care trebuie să o luați atunci când scrieți programele este dacă supraîncărcările măresc sau nu lizibilitatea programului.

ȘABLOANELE

C/C++1109

După cum ați învățat, probabil că de multe ori programele dumneavoastră trebuie să duplice o funcție astfel încât ea să accepte parametri de un tip diferit. De exemplu, următoarea funcție, *compara_valori*, compară două valori de tipul *int* și returnează valoarea mai mare:

```
int compara_valori(int a, int b)
  {
    return ((a > b) ? a: b);
  }
```

Dacă programul dumneavoastră trebuie mai apoi să compare două valori reale, va trebui să creați o a doua funcție. Cea de-a doua funcție efectuează aceleași prelucrări, dar acceptă tipuri diferite, după cum vedeți mai jos:

```
float compara_valori(float a, float b)
  {
    return ((a > b) ? a: b);
  }
```

Dacă cele două funcții apar în același program, va trebui fie să supraîncărcați funcția inițială (creând o posibilitate de confuzie) sau să selectați un unic nume pentru fiecare dintre funcții. Pentru a vă ajuta să reduceți numărul unor astfel de definiții de funcții duplicate și a face programele dumneavoastră mai ușor de înțeles, limbajul C++ acceptă definirea de *șabloane*. Un șablon pune la dispoziție formatul funcțiilor și substituenții de tip. Următorul fragment de cod arată formatul general al unui șablon de funcție, unde *T* este tipul pe care compilatorul îl va înlocui ulterior:

```
template<class T> T nume_functie(T param_a, T param_b)
  {
    // Instructiuni
  }
```

Ca exemplu, iată următorul șablon de funcție pentru funcția *compara_valori*:

```
template<class T> T compara_valori(T a, T b)
  {
    return ((a > b) ? a: b);
  }
```

După cum puteți vedea, compilatorul poate înlocui litera *T* cu oricare dintre tipurile *int* sau *float* pentru a crea funcțiile arătate mai înainte. Câteva dintre secțiunile care urmează vor explica în detaliu șabloanele.

***Observație:** Compilatorul **Turbo C++ Lite** aflat pe CD-ROM-ul care însoțește această carte nu acceptă definițiile generice. Pentru a scrie programe care utilizează definiții generice, trebuie să alegeți un alt compilator, cum ar fi Borland C++ 5.02 sau Microsoft **Visual C++**.*

1110 UTILIZAREA UNUI ȘABLON SIMPLU



După cum ați învățat în secțiunea 1109, limbajul C++ acceptă utilizarea funcțiilor șablon. Următorul program, *compara.cpp*, utilizează funcția șablon *compara_valori* pentru a compara valori de tipuri diferite:

```
#include <iostream.h>
template<class T> T compara_valori(T a, T b)
  {
    return((a > b) ? a: b);
  }
float compara_valori(float a, float b);
int compara_valori(int a, int b);
long compara_valori(long a, long b);
void main(void)
```

```
{
    float a = 1.2345, b = 2.34567;
    cout << "Comparam " << a << ' ' << b << ' ' <<
          compara_valori(a, b) << endl;
    int c = 1, d = 1001;
    cout << "Comparam " << c << ' ' << d << ' ' <<
          compara_valori(c, d) << endl;
    long e = 1010101L, f = 2020202L;
    cout << "Comparam " << e << ' ' << f << ' ' <<
          compara_valori(e, f) << endl;
}
```

În exemplul precedent, șablonul de la începutul programului *compara.cpp* specifica instrucțiunile funcției și substituenții de tip. Atunci când limbajul C++ întâlnește prototipurile ce apar înainte de funcția *main*, el creează funcțiile necesare. Mai târziu, compilatorul determină care dintre funcții să o utilizeze, bazându-se pe tipul parametrilor din program, transmiși șablonului generic (*float, int* sau *long*).

FUNCȚIILE GENERICE

C/C++1111

După cum ați învățat, limbajul C++ acceptă funcțiile generice. O funcție generică definește un set general de operații pe care funcția le va aplica asupra diverselor tipuri de date. O funcție generică primește ca parametru tipul datelor pe care va opera funcția. Deoarece funcția generică nu este explicit scrisă, prin natura sa, puteți utiliza aceeași procedură generală din cadrul funcției cu un număr mare de tipuri de date. Crearea de funcții generice în programele dumneavoastră poate fi folositoare deoarece mulți algoritmi sunt practic identici în ceea ce privește prelucrările operate independent de tipul de date utilizat. În următoarele secțiuni veți învăța despre biblioteca *Standard Template Library* (*STL*), care aplică acest concept de funcții generice unei largi game de algoritmi generali. De exemplu, după cum ați învățat în secțiunile precedente, algoritmul de sortare rapidă este întotdeauna același, indiferent de tipul de date pe care îl sortează programul dumneavoastră. Deoarece fișierul antet a creat algoritmul de sortare rapidă ca pe o funcție generică, nu mai este nevoie să creați mai multe funcții care efectuează aceeași prelucrare de bază.

În cadrul programelor dumneavoastră, veți utiliza cuvântul cheie *template* pentru a crea funcții generice. În limba engleză termenul „template" înseamnă „șablon", fiind deci potrivit pentru scopul funcțiilor generice. Cu alte cuvinte veți utiliza funcțiile generice pentru a crea un șablon de prelucrare, pe care programul dumneavoastră îl va utiliza cu informații specifice. Forma generală a unei declarații de funcție generică este arătată mai jos:

```
template <class Ttip> tip-retur nume_functie([lista de parametri])
  {
    // corpul functiei
  }
```

Ttip este un substituent al numelui tipului de date pe care îl va utiliza funcția. Puteți de asemenea să utilizați numele *Ttip* în cadrul definiției funcției. Cu alte cuvinte, *Ttip* este un substituent pe catre compilatorul îl va înlocui în mod automat cu tipul de date, de fiecare dată când se creează o versiune specifică a funcției. În programul *compara.cpp*, detaliat în

secțiunea 1110, de exemplu, compilatorul va înlocui funcția generică cu trei funcții specifice: una care returnează o valoare de tipul *float*, alta care returnează o valoare de tipul *int* și alta care returnează o valoare de tipul *long*. În secțiunea 1112 veți învăța despre șabloanele care acceptă mai multe tipuri.

1112 ȘABLOANE CARE ACCEPTĂ MAI MULTE TIPURI

C/C++

Atunci când lucrați cu șabloane în C++, uneori șablonul va solicita mai multe tipuri de date. De exemplu, să considerăm următoarea funcție, *ad_valori* care adună o valoare de tipul *long* și una de tipul *int* și returnează un rezultat de tipul *long*:

```
long ad_valori(long a, int b);
  {
    return(a + b);
  }
```

Pentru a crea un șablon pentru funcția *ad_valori*, trebuie să specificați două tipuri (cum sunt T și T1), ca mai jos:

```
template<class T, class T1> T ad_valori(T a, T1 b)
  {
    return(a + b);
  }
```

În exemplul precedent, compilatorul va substitui fiecare apariție a clasi *T* cu tipul pe care îl specificați. La fel, compilatorul va substitui fiecare apariție a clasei *T1* cu clasa corespunzătoare din apelul dumneavoastră.

În exemplul următor, compilatorul va substitui fiecare apariție a clasei *T* cu tipul pe care îl specificați. La fel, compilatorul va substitui fiecare apariție a clasei *T1* cu clasa corespunzătoare din apelul dumneavoastră. Următorul program, *sablon.cpp* utilizează șablonul *ad_valori*:

```
#include <iostream.h>

template<class T, class T1> T ad_valori(T a, T1 b)
  {
    return(a + b);
  }

long ad_valori(long a, int b);
double ad_valori(double a, float b);

void main(void)
  {
    long a = 320000L;
    int b = 31000;

    double c = 22.0 / 7.0;
    float d = 3.145;

    cout << "Adunam " << a << ' ' << b << ' ' << ad_valori(a, b)
```

```
        <<endl;
    cout << "Adunam " << c << ' ' << d << ' ' << ad_valori(c, d)
        <<endl;
}
```

Când compilați și executați programul *sablon.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Adunam 320000 31000 351000
Adunam 3.14286
3.145 6.28786
C:\>
```

MAI MULTE DESPRE ȘABLOANELE CARE ACCEPTĂ MAI MULTE TIPURI GENERICE

C/C++1113

După cum ați învățat, puteți declara șabloane care să accepte mai multe tipuri. În secțiunea 1112, de exemplu, ați creat o funcție care acceptă două tipuri și returnează rezultatul de primul tip. Însă funcțiile dumneavoastră generice pot accepta un număr nelimitat de tipuri generice. Forma generală a funcțiilor generice ce acceptă tipuri multiple este arătată mai jos:

```
template <class Ttip1, class Ttip2, … class TtipN>
        tip_retur nume_functie([lista parametrilor])
    {
      // corpul functiei
    }
```

Atunci când declarați o funcție generică ce acceptă tipuri multiple, trebuie să evitați să declarați prea multe tipuri generice în cadrul funcției, deoarece aceasta poate produce mai mult confuzie decât soluții. De asemenea, trebuie să vă asigurați că funcția poate deduce întotdeauna tipul *tip_retur* din tipurile generice pe care le primește ca parametri.

SUPRAÎNCĂRCAREA EXPLICITĂ A UNEI FUNCȚII GENERICE

C/C++1114

În secțiunile precedente ați învățat cum să definiți funcțiile generice. De asemenea, ați învățat că o funcție generică este prin esență „auto-încărcabilă"- cu alte cuvinte, ea creează atâtea versiuni ale ei însăși câte sunt necesare. Totuși, probabil că uneori veți avea nevoie să supraîncărcați în mod explicit un șablon. De exemplu, puteți să aveți o funcție *suma* generalizată, care acționează diferit atunci când adună valori de tip *long*. Dacă supraîncărcați o funcție generică, funcția supraîncărcată „ascunde" funcția generică pentru instanțele cu acele valori specificate. Următorul program, *supr_sab.cpp* supraîncarcă o explicit o funcție șablon:

```
#include <iostream.h>

template <class X> void comuta(X &a, X &b);
void comuta(int &a, int &b);

void main(void)
{
    int i=10, j=20;
```

```
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "i, j initiale: " << i << " " << j << endl;
    cout << "x, y initiale: " << x << " " << y << endl;
    cout << "a, b initiale: " << a << " " << b << endl;
    comuta(i,j); // supraincarcare explicita a functiei comuta
    comuta(a,b);
    comuta(x,y);
    cout << "i, j inversate: " << i << " " << j << endl;
    cout << "x, y inversate: " << x << " " << y << endl;
    cout << "a, b inversate: " << a << " " << b << endl;
  }
template <class X> void comuta(X &a, X &b)
  {
    X temp;
    temp = a;
    a = b;
    b = temp;
  }
void comuta(int &a, int &b)
  {
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Acum ne aflam in functia comuta supraincarcata."
      << endl;
  }
```

Atunci când compilați și executați programul *supr_sab.cpp*, ecranul dumneavoastră va afișa următoarele:

```
i,j initiale: 10 20
x,y initiale: 10.1 23.3
a,b initiale: x z
Acum ne aflam in functia comuta supraincarcata
i,j inversate: 20 10
x,y inversate: 23.3 10.1
a,b inversate: z x
C:\>
```

1115 RESTRICȚIILE ASUPRA FUNCȚIILOR GENERICE



După cum ați învățat, funcțiile generice sunt similare funcțiilor supraîncărcate. Însă, pe lângă puterea funcțiilor generice, limbajul C++ aplică de asemenea mai multe restricții acestor funcții generice decât asupra celor supraîncărcate. Atunci când supraîncărcați funcții, ele pot efectua diverse acțiuni, în funcție de criteriul de supraîncărcare. Însă, atunci când scrieți o

funcție generică, funcția dumneavoastră trebuie să efetueze aceeași prelucrare asupra tuturor datelor, indiferent de tip. De exemplu, în următorul program, *nu_sab.cpp*, nu puteți înlocui funcțiile supraîncărcate cu o funcție generică deoarece activitățile funcțiilor supraîncărcate sunt diferite:

```
#include <iostream.h>
#include <math.h>
void nu_sablon(int i)
  {
    cout << "valoarea este: " << i << endl;
  }
void nu_sablon(double d)
  {
    double parteint;
    double partefrac;
    partefrac = modf(d, &parteint);
    cout << "Partea fractionara: " << partefrac << endl;
    cout << "Partea intreaga: " << parteint << endl;
  }
void main(void)
  {
    nu_sablon(1);
    nu_sablon(12.2);
  }
```

UTILIZAREA UNEI FUNCȚII GENERICE

C/C++1116

După cum puteți vedea, una dintre cele mai importante facilități ale limbajului C++ sunt funcțiile generice. Puteți aplica funcțiile generice tuturor tipurilor de situații. De fiecare dată când aveți o funcție ce definește un algoritm ce poate fi generalizat, puteți face din această funcție o funcție șablon. După ce ați creat funcția generică, o puteți utiliza cu orice tip de date, fără a mai fi nevoie să rescrieți funcția. În următoarele secțiuni veți învăța cum să utilizați șabloane pentru a defini clase. În secțiunea 1117 veți crea o funcție generică de sortare cu metoda bulelor (*bubble sort*). Pentru început, asigurați-vă că ați înțeles bine modul de funcționare al funcțiilor generice înainte de a trece la secțiunile următoare. Pentru a înțelege mai bine funcțiile generice, analizați următorul program simplu, *arata_txt.cpp*, care definește o funcție generică de afișare pe ecran:

```
#include <iostream.h>
template <class T1, class T2> void exemplu(T1 x, T2 y);
void main(void)
  {
    exemplu(10, "hi");
    exemplu(0.23, 10L);
    exemplu("Jamsa's", "C/C++");
```

```
}
template <class T1, class T2> void exemplu(T1 x, T2 y)
  {
    cout << x << " " << y << endl;
  }
```

Atunci când compilaţi şi executaţi programul *arata_txt.cpp*, compilatorul va crea trei versiuni ale funcţiei *exemplu*: una care acceptă un întreg şi un şir de caractere, una care acceptă o valoare *float* şi una *long* şi una care acceptă două şiruri de caractere. Însă, atunci când executaţi programul *arata_txt.cpp*, faptul că prin compilator s-au creat trei versiuni ale funcţiei (în loc ca programul însuşi să fi conţinut trei versiuni ale funcţiei) nu este vizibil pentru utilizator, care va vedea doar următoarea ieşire pe ecran:

```
10 hi
0.23 10
Jamsa's C/C++
C:\>
```

Pentru a efectua aceiaşi paşi în absenţa unei funcţii generice, ar fi trebuit să scrieţi un program ca *arata_nu.cpp*, arătat mai jos:

```
#include <iostream.h>

void exemplu(int x, char *y);
void exemplu(float x, long y);
void exemplu(char *x, char *y);

void main(void)
  {
    exemplu(10, "hi");
    exemplu(0.23, 10L);
    exemplu("Jamsa's", "C/C++");
  }

void exemplu(int x, char *y)
  {
    cout << x << " " << y << endl;
  }

void exemplu(float x, long y)
  {
    cout << x << " " << y << endl;
  }

void exemplu(char *x, char *y)
  {
    cout << x << " " << y << endl;
  }
```

UTILIZAREA UNEI FUNCȚII GENERICE DE SORTARE PRIN METODA BULELOR

C/C++1117

După cum ați învățat, funcțiile generice pot fi de folos în a vă face programele mai ușor de înțeles și economisesc timpul afectat programării. Pentru a înțelege mai bine utilitatea funcțiilor generice, analizați un program pe care înainte l-ați scris fără funcții generice și veți vedea cum face o funcție generică ca acest program să fie mai util. De exemplu, în secțiunile 492 și 493, ați creat o funcție simplă de sortare prin metoda bulelor care lucra doar cu valori întregi. Aceasta funcție de sortare ar fi cu mult mai semnificativă dacă ar fi fost utilizat un șablon generic și nu o funcție specifică. Funcția inițială care accepta o matrice de întregi pentru a fi sortată a fost scrisă ca mai jos:

```
void sortare_bule(int matrice[], int dim)
  {
    int temp, i, j;

    for (i = 0; i < dim; i++)
      for (j = 0; j < dim; j++)
        if (matrice[i] < matrice[j])
          {
            temp = matrice[i];
            matrice[i] = matrice[j];
            matrice[j] = temp;
          }
  }
```

Cu ajutorul cunoștințelor dumneavoastră despre funcții generice, puteți scrie acum codul pentru sortarea prin metoda bulelor astfel încât să acopere toate tipurile de valori, nu numai matrice de tip *int*, după cum arătăm în programul *bule.cpp*:

```
#include <iostream.h>

template <class X> void sortare_bule(X *elemente, int dim)
template <class X> void arata_elemente(X *elemente, int dim)

void main(void)
  {
    int tablouint[7] = {7, 5, 4, 3, 9, 8, 6};
    double tabloud[5] = {4.2, 2.5, -0.9, 100.2, 3.0};

    cout << "Matrice de valori intregi neordonata: " << endl;
    arata_elemente(tablouint, 7);
    cout << " Matrice de valori double neordonata: " << endl;
    arata_elemente(tabloud, 5);
    sortare_bule(tablouint, 7);
    sortare_bule(tabloud, 5);
    cout << " Matrice de valori intregi ordonata: " << endl;
    arata_elemente(tablouint, 7);
    cout << " Matrice de valori double ordonata: " << endl;
    arata_elemente(tabloud, 5);
  }
```

```
template <class X> void sortare_bule(X *elemente, int dim)
  {
    register int i, j;
    X temp;

    for (i = 1; i < dim; i++)
      for (j = dim-1; j >= i; j--)
        if (elemente[j-1] > elemente[j])
          {
            temp = elemente[j-1];
            elemente[j-1] = elemente[j];
            elemente[j] = temp;
          }
  }

template <class X> void arata_elemente(X *elemente, int dim)
  {
    int i;
    for(i=0; i < dim; i++)
      cout << elemente[i] << ", ";
    cout << endl;
  }
```

Atunci când compilați și executați programul *bule.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Matrice de valori intregi neordonata:
7, 5, 4, 3, 9, 8, 6,
Matrice de valori double neordonata:
4.2, 2.5, -0.9, 100.2, 3,
Matrice de valori intregi ordonata:
3, 4, 5, 6, 7, 8, 9,
Matrice de valori double ordonata:
-0.9, 2.5, 3, 4.2, 100.2,
C:\>
```

1118 UTILIZAREA FUNCȚIILOR GENERICE PENTRU COMPACTAREA UNEI MATRICE



Atunci când programele dumneavoastră lucrează cu matrice, puteți să eliminați elemente din mijlocul matricei și să mutați elementele rămase în matrice „în jos", pentru a umple spațiul liber creat – proces denumit *compactare*. O modalitate de a evita necesitatea compactării matricelor este utilizarea unei liste înlănțuite, despre care ați învățat anterior. Dar deseori vi se va părea mai convenabil să lucrați cu o matrice. Următorul program, *compact.cpp* utilizează o funcție generică pentru a compacta matrice de diverse tipuri:

```
#include <iostream.h>

template <class X> void compact(X *elemente, int nr, int start,
                               int final)
template <class X> void arata_elemente(X *elemente, int dim)
```

```
void main(void)
  {
    int numere[7] = {0, 1, 2, 3, 4, 5, 6};
    char sir[17] = "Functii generice";

    cout << "Matrice de valori intregi necompactata: ";
    arata_elemente(numere, 7);
    cout << "Matrice de caractere necompactata: ";
    arata_elemente(sir, 17);
    compact(numere, 7, 2, 4);
    compact(sir, 17, 6, 10);
    cout << "Matrice de valori intregi compactata: ";
    arata_elemente(numere, 7);
    cout << "Matrice de caractere compactata: ";
    arata_elemente(sir, 17);
  }
template <class X> void compact(X *elemente, int nr, int start,
                                 int final)
  {
    register int i;
    for(i=final+1; i < nr; i++, start++)
      elemente[start] = elemente[i];
    for( ; start<nr; start++)
      elemente[start] = (X) 0;
  }
template <class X> void arata_elemente(X *elemente, int dim)
  {
    int i;
    for(i=0; i < dim; i++)
      cout << elemente[i];
    cout << endl;
  }
```

În exemplul precedent, funcția *compact* completează elementele rămase din matrice cu valoarea zero. Atunci când compilați și executați programul *compact.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Matrice de valori intregi necompactata: 0123456
Matrice de caractere necompactata: Functii generice
Matrice de valori intregi compactata: 0156000
Matrice de caractere compactata: Functierice
C:\>
```

UNDE PLASĂM ȘABLOANELE

C/C++1119

După cum ați învățat, șabloanele limbajului C++ vă permit reducerea programării de funcții care diferă doar în privința parametrilor și tipurilor returnate. Atunci când creați șabloane, ar trebui să le plasați în cadrul unor fișiere antet cu o denumire semnificativă, pentru a le putea

reutiliza uşor în alte programe. Deoarece iniţial veţi crea probabil numai câteva şabloane, aţi putea utiliza fişierul antet *sabloane.h*. Pe măsură ce veţi crea mai multe şabloane, le veţi putea plasa în alte fişiere antet, în funcţie de utilitatea lor.

1120 ŞABLOANELE ELIMINĂ ŞI CLASELE DUPLICATE

C/C++

După cum aţi învăţat, şabloanele limbajului C++ vă permit reducerea programării de funcţii care diferă doar în privinţa parametrilor şi tipurilor returnate. Programele dumneavoastră pot utiliza de asemenea şabloane pentru a elimina clasele similare. De exemplu, să considerăm următoarele clase:

```
class DistantaShort {
   public:
     DistantaShort(int distanta) { DistantaShort::distanta =
       distanta; };
     ArataDistanta(void) { cout << "Distanta este " << distanta
       << " mile" << endl; };
   private:
     int distanta;
};

class DistantaLong {
   public:
     DistantaLong(int distanta) { DistantaLong::distanta =
       distanta; };
     ArataDistanta(void) { cout << "Distanta este " << distanta
       << " mile" << endl; };
   private:
     long distanta;
};
```

Ambele clase efectuează prelucrări similare, unica diferenţă fiind faptul că *DistantaLong* manevrează valori de tipul *long int*, iar *DistantaShort* doar valori de tipul *int*. Deoarece clasele sunt identice în fond şi execută aceleaşi procese, diferenţa constând doar în tipurile valorilor conţinute, aceste clase sunt candidate recomandate pentru a forma o singură clasă generică.

Următorul program, *classab.cpp* combină cele două tipuri de clase, *DistantaShort* şi *DistantaLong* într-o singură clasă generică, *Distanta*:

```
#include <iostream.h>

template<class T> class Distanta {
   public:
     Distanta(T distanta);
     void arata_distanta(void) {
        cout << "Distanta este de " << distanta << " de mile\n"; }
   private:
     T distanta;
```

```
};
template<class T>
Distanta<T>::Distanta(T distanta) { Distanta::distanta =
  distanta; };
void main(void)
  {
    Distanta<int> distanta_short(100);
    Distanta<long> distanta_long(2000000L);
    distanta_short.arata_distanta();
    distanta_long.arata_distanta();
  }
```

Atunci când compilați și executați programul *classab.cpp*, compilatorul va crea clasele utilizând tipurile corecte. După cum veți învăța în secțiunea 1121, șabloanele de clase cum ar fi *Distanta* sunt deseori denumite *clase generice* sau *generatori de clase*. Atunci când executați programul *classsab.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Distanta este de 100 de mile
Distanta este de 2000000 de mile
C:\>
```

CLASELE GENERICE

C/C++1121

După cum ați învățat în secțiunea 1120, limbajul C++ vă permite să utilizați șabloane pentru definirea claselor generice. Atunci când specificați un șablon de clasă, trebuie să specificați întotdeauna numele corespunzător, urmat de paranteze unghiulare și un tip (fie un tip care va fi substituit, fie tipul real), ca mai jos:

```
Distanta<T>
Distanta<int>
```

Examinând programele în C++ care utilizează șabloane de clase, veți întâlni unele programe care definesc șabloane ce acceptă parametri, ca în exemplul de mai jos:

```
template<class T, int dim_matrice = 64> class Oclasa {
    // Instructiuni
};
```

În acest caz, șablonul nu specifică numai un substituent de tip, ci și un parametru pe care programul îl poate utiliza în cadrul acelui șablon. Atunci când programul va utiliza mai târziu șablonul, el îi va putea transmite o valoare ca parametru, după cum arătăm mai jos:

```
OClasa<int, 1024> aceasta_instanta;
```

UTILIZAREA CLASELOR GENERICE

C/C++1122

În secțiunile precedente ați creat și utilizat o serie de funcții generice. Însă, după cum ați învățat în secțiunea 1121, limbajul C++ acceptă și clase generice. Crearea unei clase generice, deși adesea mai dificilă decât crearea unei clase obișnuite, mărește de asemenea

puterea programelor dumneavoastră. Puteți utiliza clase generice pentru a permite programelor dumneavoastră să acceseze un set considerabil mai larg de tipuri de valori fără a fi nevoie de semnificativ mai mult cod. După cum veți învăța în următoarele secțiuni, *biblioteca standard de șabloane* (*Standard Template Library - STL*) este construită în jurul conceptului de clase generice. Pentru a înțelege mai bine clasele generice, analizați următorul program, *gen_stv.cpp*:

```
#include <iostream.h>

const int DIM = 100;

template <class STip> class stiva {
    STip stv[DIM];
    int vfs;
public:
    stiva(void);
    ~stiva(void);
    void depune(STip i);
    STip extrage(void);
};

template <class STip> stiva<STip>::stiva()
   {
    vfs = 0;
    cout << "Stiva initializata." << endl;
   }

template <class STip> stiva<STip>::~stiva()
   {
    cout << "Stiva distrusa." << endl;
   }

template <class STip> void stiva<STip>::depune(STip i)
   {
    if(vfs==DIM)
      {
       cout << "Stiva este plina." << endl;
       return;
      }
    stv[vfs++] = i;
   }
template <class STip> STip stiva<STip>::extrage(void)
   {
    if(vfs==0)
      {
       cout << "Stiva depasita in jos." << endl;
       return 0;
      }
    return stv[--vfs];
   }
```

```
void main(void)
  {
    stiva<int> a;
    stiva<double> b;
    stiva<char> c;
    int i;

    a.depune(1);
    a.depune(2);
    b.depune(99.3);
    b.depune(-12.23);

    cout << a.extrage() << " ";
    cout << a.extrage() << " ";
    cout << b.extrage() << " ";
    cout << b.extrage() << endl;

    for(i=0; i<10; i++)
      c.depune((char) 'A' + i);
    for(i=0; i<10; i++)
      cout << c.extrage();
    cout << endl;
  }
```

Programul *gen_stv.cpp* definește clasa generică *stiva*, care lucrează cu o matrice de 100 de elemente de tipul definit și o valoare întreagă pe care o puteți utiliza pentru a accesa elementele din interiorul stivei. După cum ați văzut în definițiile precedente ale clasei *stiva*, sunt definite și funcțiile membre *depune* și *extrage* pe care le puteți utiliza pentru a plasa și a extrage date dintr-un obiect *stiva*. Atunci când programul își începe execuția, el creează trei instanțe ale clasei generice *stiva*: o stivă *int*, o stivă *double* și una *char*. Restul proceselor programului manevrează fiecare stivă pe rând. Atunci când compilați și executați programul *gen_stv.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Stiva initializata
Stiva initializata
Stiva initializata
2 1 -12.23 99.3
JIHGFEDCBA
Stiva distrusa
Stiva distrusa
Stiva distrusa
C:\>
```

CREAREA UNEI CLASE GENERICE CU DOUĂ TIPURI GENERICE

C/C++1123

După cum ați învățat, programele dumneavoastră pot declara funcții generice ce acceptă mai multe tipuri de date generice. În mod similar, programele pot declara clase generice care să accepte mai multe tipuri de date generice. Pentru a declara o clasă generică ce acceptă tipuri de date generice multiple, veți utiliza o formă extinsă a declarației standard a unei clase generice, ca mai jos:

```
template<class T1, class T2, … class TN> class nume_clasa {
    // definitiile membrilor
}
```

Înlocuitorii de tip *T1,T2,* până la *TN* reprezintă tipurile generice pe care le vor accepta clasele dumneavoastră. O clasă poate, teoretic, să accepte un număr infinit de tipuri generice în cadrul deiniției sale. Totuși, ca și în cazul funcțiilor generice ce acceptă tipuri multiple, trebuie de asemenea să aveți grijă și să nu definiți prea multe tipuri generice în clasele dumneavoastră, fiindcă astfel creați confuzie în programe. Pentru a înțelege mai bine cum veți declara clasele generice care acceptă două sau mai multe tipuri de date generice, studiați programul *doua_gen.cpp*, arătat mai jos:

```
#include <iostream.h>

template <class T1, class T2> class doua_gen {
    T1 i;
    T2 j;
public:
    doua_gen(T1 a, T2 b)
      { i=a; j=b;}
    void arata(void)
      { cout << i << " " << j << endl; }
};

void main(void)
  {
    doua_gen<int, double> ob1(10, 0.23);
    doua_gen<char, char *> ob2('X', "Acesta este un test.");
    ob1.arata();
    ob2.arata();
  }
```

1124 CREAREA UNUI MANIPULATOR PARAMETRIZAT

C/C++

După cum ați învățat în secțiunea 1001, veți utiliza clasele generice pentru a crea manipulatori personalizați. Atunci când creați un manipulator parametrizat, trebuie să includeți fișierul *iomanip.h* în programul dumneavoastră. Fișierul *iomanip.h* definește trei clase generice: *omanip, imanip* și *smanip.* În general, atunci când trebuie să creați un maipulator ce primește un argument, programul dumneavoastră trebuie să creeze două funcții manipulator supraîncărcate. Prima trebuie să definească doi parametri: o referință la flux și un parametru pe care prima funcție îl transmite celei de-a doua funcții (funcția generică). Funcția generică acceptă un singur parametru și generează un apel către cea dintâi funcție. Pentru a înțelege mai bine relațiile dintre cele două funcții, studiați forma generală a manipulatorului parametrizat de ieșire, arătată mai jos:

```
ostream &nume-manip(ostream &flux,tip param)
  {
```

```
        // aici va fi codul dumneavoastra
        return flux;
    }
// supraincarcarea functiei generice
omanip<tip> nume-manip(tip param)
    {
        return omanip<tip> (nume-manip, param);
    }
```

În forma generalizată, *nume-manip* este numele manipulatorului, iar *tip* specifică tipul parametrului pe care-l manevrează manipulatorul. Deoarece *omanip* este de asemenea o clasă generică, *tip* este de asemenea tipul de date asupra căruia acționează obiectul *omanip*(pe care îl returnează cea de-a doua funcție supraîncărcată).

Cu siguranță, scrierea de manipulatori parametrizați este o sarcină difiiclă și adesea confuză. Pentru a înțelege mai bine modul în care veți utiliza manipulatorii parametrizați într-un program, studiați programul *parm_man.cpp*, arătat mai jos:

```
#include <iostream.h>
#include <iomanip.h>

ostream &indent(ostream &flux, int lungime)
    {
        register int i;

        for(i=0; i<lungime; i++)
          cout << " ";
        return flux;
    }

omanip<int> indent(int lungime)
    {
        return omanip<int>(indent, lungime);
    }

void main(void)
    {
        cout << indent(10) << "Acesta este un test" << endl;
        cout << indent(20) << "al noului manipulator indent." << endl;
        cout << indent(5) << "Merge!" << endl;
    }
```

Atunci când compilați și executați programul *parm_manip.cpp*, ecranul dumneavoastră va afișa următoarea ieșire:

```
          Acesta este un test
                    al noului manipulator indent.
     Merge!
C:\>
```

1125 CREAREA UNEI CLASE MATRICE GENERICĂ C/C++

În secțiunile precedente ați învățat despre metodele cu care puteți crea clase și funcții generice și activități pe care șabloanele vi le permit sau vă ajută să le efectuați. După cum ați văzut în multe dintre secțiunile precedente ale acestei cărți, în majoritatea programelor veți crea o matrice de clase. Prin urmare, trebuie să înțelegeți cum veți manipula o clasă generică pentru a inițializa o matrice de această clasă.

Crearea unei matrice generice este la fel de simplă ca și crearea unei matrice de un tip normal. De exemplu, dacă toate matricele dumneavoastră sunt de aceeași dimensiune, ar trebui să utilizați o clasă generică fără parametri pentru a declara o matrice de întregi, ca mai jos:

```
untip<int> tablou_int;
```

Utilizarea unei clase generice pentru a crea matrice vă permite să operați asupra lor în mod diferit. După cum ați învățat înainte, programele dumneavoastră pot supraîncărca operatorul [] pentru a crea „matrice sigure" – matrice ale căror limite nu pot fi depășite de programe, pe lângă alte criterii de siguranță a matricelor. Când creați o clasă generică și supraîncărcați operatorul [ ] în raport cu această clasă, puteți forța orice matrice din cadrul programului dumneavoastră să fie „sigură". Următorul program, *gen_sig.cpp* utilizează o clasă generică și un operator supraîncărcat [ ] pentru a crea matrice „sigure":

```
#include <iostream.h>
#include "stdlib.h"

const int DIM = 10;

template <class UnTip> class UnTip {
    UnTip a[DIM];
public:
    UnTip(void)
      {
       int i;
       for(i=0; i<DIM; i++)
         a[i] = i;
      }
    UnTip &operator[](int i);
};

template <class UnTip> UnTip &UnTip<UnTip>::operator[](int i)
    {
     if(i<0 || i> DIM-1)
      {
       cout << endl << "Valoarea indicelui";
       cout << i << " este in afara limitelor." << endl;
      }
     return a[i];
    }

void main(void)
```

```
{
  UnTip<int> int_tablou;
  UnTip<double> double_tablou;
  int i;

  cout << "Matrice de intregi: ";
  for(i=0; i<DIM; i++)
   int_tablou[i] = i;
  for(i=0; i<DIM; i++)
   cout << int_tablou[i] << " ";
  cout << endl;

  cout << "Matrice de double: ";
  cout.precision(2);
  for(i=0; i<DIM; i++)
   double_tablou[i] = (double)i/3;
  for(i=0; i<DIM; i++)
   cout << double_tablou[i] << " ";
  cout << endl;

  int_matrice[12] = 100; // Apeleaza operatorul matrice
                         // supraincarcat
}
```

Programul *gen_sig.cpp* creează clasa generică *untip* și supraîncarcă operatorul [ ] în cadrul clasei. Funcția supraîncărcată se asigură că utilizatorul nu încearcă să acceseze sau să inițializeze vreun element al matricei din afara limitelor predefinite ale matricei. Atunci când programul *gen_sig.cpp* se execută, el creează două matrice (una de tip *int* și alta de tip *double*), completează matricele și afișează valorile lor. În sfârșit, ultima instrucțiune încearcă să inițializeze o valoare din afara limitelor matricei. Atunci când compilați și executați programul *gen_sig.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Matrice de intregi: 0 1 2 3 4 5 6 7 8 9
Matrice de double: 0 0.33 0.67 1 1.3 1.7 2 2.3 2.7 3

Valoarea indicelui 12 este in afara limitelor.
C:\>
```

TRATAREA EXCEPȚIILOR

C/C++1126

Pe măsură ce creați biblioteci de clase, uneori puteți anticipa tipurile de erori la momentul execuției pe care programul dumneavoastră le va întâlni pe parcursul lucrului cu clasa (cum ar fi depășirea limitelor matricei sau transmiterea unei valori prea mari). Din păcate, de multe ori se va întâmpla să nu puteți scrie codul care să detecteze astfel de erori care apar în alte programe. Pentru aceste cazuri, majoritatea compilatoarelor de C++ acceptă *programe handler pentru excepții*. În general, un *handler pentru excepții* este un cod ce se execută atunci când apare o eroare. În cadrul codului bibliotecilor de clase, veți testa diferitele erori ce pot apărea. Dacă are loc o eroare, veți iniția o excepție (în C++ aceasta se numește throw – a lansa). Programul care a suferit acea eroare este responsabil pentru captarea și tratarea erorii - ceea ce înseamnă că programul trebuie să prevadă codul care tratează eroarea. Pentru acceptarea tratării excepției, limbajul C++ pune la dispoziție cuvintele cheie arătate în tabelul 1126.

Cuvântul cheie	Semnificație
catch	Captează excepția inițiată
throw	**Inițiaza un handler de eroare**
try	**Încearcă o operație care să testeze o posibilă excepție**

Tabelul 1126 *Cuvintele cheie ale limbajului C++ referitoare la tratarea excepțiilor.*

1127 FORMA DE BAZĂ A TRATĂRII EXCEPȚIILOR C/C++

După cum ați învățat în secțiunea 1126, limbajul C++ vă pune la dispoziție câteva instrucțiuni pe care programele dumneavoastră le pot utiliza pentru a capta excepțiile din cadrul programelor dumneavoastră. Cele trei componente de bază ale fiecărui *handler de excepții* sunt instrucțiunile *try*, *catch* și *throw*. Atunci când programele dumneavoastră efectuează prelucrarea excepțiilor, trebuie să includeți în cadrul unui bloc *try* instrucțiunile pe care doriți să le monitorizați în vederea unei excepții (adică erori). Dacă o instrucțiune se efectuează eronat, trebuie să lansați o eroare corespunzătoare acțiunii funcției. Programul plasează instrucțiunea *throw* în cadrul blocului *catch*, care efectuează tratarea propriu-zisă a erorii. Forma generalizată a blocului ce tratează excepțiile este:

```
try {
    // blocul try
    // if(eroare) throw valoare_exceptie;
  }
catch(tip-exceptie nume-variabila) {
    // prelucrarea exceptiei
  }
```

În cadrul formei generalizate a unui handler de excepție, valoarea *valoare_exceptie* lansată trebuie să corespundă tipului *tip-exceptie* captare, după cum veți vedea în secțiunile ce urmează. În secțiunea 1128 veți scrie un *handler de eroare* simplu, bazat pe forma generalizată arătată în această secțiune.

1128 SCRIEREA UNUI MANIPULATOR DE EXCEPȚII SIMPLU C/C++

După cum ați învățat în secțiunea 1127, fiecare handler de excepții pe care îl scrieți în programele dumneavoastră va conține o instrucțiune *try*, una sau mai multe instrucțiuni *throw* și una sau mai multe instrucțiuni *catch*. Pentru a înțelege mai bine pocesările pe care le efectuează un handler de excepții, studiați următorul program, *simpla_e.cpp* care utilizează un handler de excepții pentru lansarea și tratarea unei erori din cadrul funcției *main*:

```
#include <iostream.h>

void main(void)
  {
    cout << "Start" << endl;
    try {
      cout << "In interiorul blocului try." << endl;
      throw 100;
```

```
        cout << "Nu se va executa.";
      }
      catch(int i) {
        cout << "Am captat o exceptie -- valoarea este: ";
        cout << i << endl;
      }
      cout << "Sfarsit";
   }
```

Programul *simpla_e.cpp* implementează un bloc *try-catch* simplu. În loc să aștepte ca programul să comită o eroare, el utilizează instrucțiunea *throw* (despre care veți învăța în secțiunea 1129) pentru producerea unei erori. După ce blocul *try* lansează eroarea, blocul *catch* o captează și prelucrează valoarea transmisă de instrucțiunea *throw*. Atunci când compilați și executați programul *simpla_e.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Start
In interiorul blocului try.
Am captat o exceptie -- valoarea este: 100
Sfarsit
C:\>
```

INSTRUCȚIUNEA THROW

C/C++1129

După cum ați învățat, cea de-a treia componentă a unui *handler de excepții* este instrucțiunea *throw* care invocă instrucțiunea *catch*. Trebuie să includeți cel puțin o instrucțiune *throw* într-un bloc *try* pentru ca programul handler de excepții să efectueze o prelucrare adecvată. Așa cum ați văzut în secțiunile precedente, formatul general al instrucțiunii *throw* este următorul:

```
throw exceptie;
```

După cum ați învățat, *exceptie* trebuie să fie unul dintre *tipurile de excepție* specificate de blocul *catch*. Pe măsură de programele dumneavoastră devin mai complexe și în mod special atunci când programați mai mult în Windows, veți observa că puteți utiliza blocurile *try-catch* (deseori numite simplu blocuri *try*) cu multe funcții, pentru a realiza programe mai stabile și mai sigure.

EXCEPȚIILE SUNT SPECIFICE TIPURILOR

C/C++1130

După cum ați învățat, programele dumneavoastră pot efectua activități de tratare a excepțiilor utilizând blocul *try*. Primul bloc *try* pe care l-ați văzut (în secțiunea 1127) capta o excepție de tip întreg. Însă, instrucțiunile dumneavoastră *catch* pot capta orice tip de excepție. Ca urmare, trebuie să aveți grijă ca excepția pe care programul dumneavoastră o lansează în cadrul blocului *try* să se potrivească cu tipul specificat de blocul *catch*. De exemplu, următorul program, *catch_d.cpp*, captează o excepție de tip *double*, în timp ce blocul *try* lansează o excepție de tip întreg:

```
#include <iostream.h>
void main(void)
```

```
{
  cout << "Start" << endl;
  try {
    cout << "In interiorul blocului try." << endl;
    throw 100;
    cout << "Nu se va executa.";
  }
  catch(double d) {
    cout << "Am captat o exceptie de tip double -- valoarea
            este: ";
    cout << d << endl;
  }
  cout << "Sfarsit";
}
```

În funcție de compilatorul dumneavoastră, atunci cînd veți compila și executa programul *catch_d.cpp*, este posibil să primiți un mesaj de avertizare de la compilator care vă anunță că instrucțiunea *catch* nu este accesibilă. Dacă primiți un astfel de mesaj la compilarea programelor dumneavoastră, trebuie să verificați cu atenție codul dumneavoastră pentru a descoperi eventuale erori logice cum ar fi cea din programul *catch_d.cpp*. Dacă primiți un astfel de mesaj de eroare la compilare și executați totuși programul, acesta va genera o ieșire asemănătoare cu cea a programului *catch_d.cpp*, prezentată mai jos:

```
Start
In interiorul blocului try.
Abnormal program termination.
C:\>
```

1131 LANSAREA EXCEPȚIILOR CU O FUNCȚIE DIN CADRUL BLOCULUI TRY



După cum ați învățat, programele dumneavoastră pot efectua ample activități de tratare a excepțiilor utilizând blocul *try*. Însă, programele dumneavoastră pot deseori să apeleze funcții din cadrul blocului *try*. Atunci când programele apelează funcții din cadrul blocului *try*, C++ va transmite excepția blocului *try*, în afara funcției (în cazul în care nu există un al doilea bloc *try* în interiorul funcției, după cum veți învăța în secțiunile următoare). Următorul program, *af_func.cpp* utilizează un bloc *try* în interiorul funcției *main* pentru a apela funcția *XHandler*.

```
#include <iostream.h>
void XHandler(int test)
  {
    cout << "Inauntrul functiei XHandler, test este:" << test
      << endl;
    if(test) throw test;
  }
void main(void)
  {
```

```
   cout << "Start: " << endl;
   try {
     cout << "Inauntrul blocului try." << endl;
     XHandler(1);
     XHandler(2);
     XHandler(0);
   }
   catch(int i) {
     cout << "Am captat o exceptie. Valoarea este: ";
     cout << i << endl;
   }
   cout << "Sfarsit";
 }
```

Atunci când compilați și executați programul *af_func.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Start:
Inauntrul blocului try.
Inauntrul functiei XHandler, test este:1
Am captat o exceptie. Valoarea este: 1
Sfarsit
C:\>
```

PLASAREA UNUI BLOC TRY ÎNTR-O FUNCȚIE

C/C++1132

După cum ați învățat, programele dumneavoastră pot utiliza blocuri *try* pentru a capta excepții în cursul prelucrării. În secțiunile precedente ați creat un bloc *try* simplu în cadrul funcției *main* și un bloc *try* dintr-o funcție *main* care apela în interiorul său o funcție. Însă puteți utiliza blocuri *try* locale în cadrul funcțiilor. Atunci când plasați un bloc *try* într-o funcție, limbajul C++ reinițializează blocul de fiecare dată când intrați în acea funcție. Următorul program, *fun_catc.cpp* vă arată cum să plasați blocuri *try* în cadrul funcțiilor:

```
#include <iostream.h>
void XHandler(int test)
  {
    try
      {
        if(test)
          throw test;
      }
    catch(int i)
      {
        cout << "Am captat exceptia nr: " << i << endl;
      }
  }
void main(void)
  {
```

```
    cout << "Start: " << endl;
    XHandler(1);
    XHandler(2);
    XHandler(0);
    XHandler(3);
    cout << "Sfarsit";
  }
```

Atunci când compilați și executați programul *fun_catc.cpp*, ecranul dumneavoastră va afișa următoarele (notați că funcția lansează doar trei excepții, deoarece al treilea apel, cu valoarea sa zero, este evaluat ca fals):

```
Start:
Am captat exceptia nr: 1
Am captat exceptia nr: 2
Am captat exceptia nr: 3
Sfarsit
C:\>
```

1133 CÂND SE EXECUTĂ INSTRUCȚIUNEA CATCH C/C++

După cum ați învățat, programele dumneavoastră pot utiliza secvența *try-catch* pentru a controla tratarea excepțiilor și a se proteja împotriva încheierii programului într-un mod anormal. În secțiunile precedente, ați învățat cum să captați excepții într-un bloc *catch*. Totuși, vă va îngrijora probabil faptul că programele dumneavoastră, deși nu lansează excepții, vor executa instrucțiuni dintr-un bloc *catch*. După cum reiese, programele dumneavoastră vor executa instrucțiunile în cadrul blocului *catch* numai dacă programul lansează o excepție în cadrul blocului *try* care se află imediat înainte. Următorul program, *nu_catch.cpp*, ilustrează modul în care programele dumneavoastră vor trece peste instrucțiunile din cadrul blocului *catch* dacă programul nu lansează o excepție:

```
#include <iostream.h>

void main(void)
  {
    cout << "Start" << endl;
    try
      {
        cout << "In interiorul blocului try." << endl;
        cout << "Inca in interiorul blocului try." << endl;
      }
    catch(int i)
      {
        cout << "Am captat o exceptie--valoarea este: " << endl;
        cout << i << endl;
      }
    cout << "Sfarsit";
  }
```

Atunci când compilați și executați programul *nu_catch.cpp*, acesta va omite instrucțiunile cuprinse în blocul *catch*, datorită faptului că programul nu lansează o excepție în blocul *try*. Programul *nu_catch.cpp* va afișa următoarele pe ecran:

```
Start
In interiorul blocului try.
Inca in interiorul blocului try.
Sfarsit
C:\>
```

UTILIZAREA MAI MULTOR INSTRUCȚIUNI CATCH CU UN SINGUR BLOC TRY

C/C++1134

Pe măsură ce tratările excepțiilor devin tot mai complexe, uneori este posibil ca un singur bloc *try* să lanseze excepții de mai multe tipuri. În cadrul programelor dumneavoastră, puteți să construiți un handler de excepții astfel încât să accepte captarea mai multor excepții. Atunci când trebuie să captați mai multe excepții, veți utiliza următorul format general:

```
try
  {
    // instructiuni
  }
catch(tip1)
  {
    // tratarea exceptiei
  }
catch(tip2)
  {
    // tratarea exceptiei
  }
.
.
.
catch(tipN)
  {
    // tratarea exceptiei
  }
```

Așa cum veți învăța în secțiunile următoare, instrucțiunile *catch* pot recunoaște orice tip returnat, nu numai tipurile de bază acceptate de C++. De fapt, instrucțiunile *catch* pot chiar să capteze tipuri lansate, definite de utilizator. Următorul program, *catch_3.cpp*, utilizează mai multe instrucțiuni *catch* pentru a capta câteva excepții de tipuri diferite:

```
#include <iostream.h>

void XHandler(int test)
  {
    try
      {
        if(test==0)
```

```
          throw test;
        if(test==1)
          throw "Sir de caractere";
         if(test==2)
          throw 123.23;
       }
     catch(int i)
       {
        cout << "Am captat exceptia #: " << i << endl;
       }
     catch(char *str)
       {
        cout << "Am captat exceptia de tip sir de caractere: " <<
               str << endl;
       }
     catch(double d)
       {
        cout << "Am captat exceptia #: " << d << endl;
       }
   }
void main(void)
   {
     cout << "Start: " << endl;
     XHandler(0);
     XHandler(1);
     XHandler(2);
     cout << "Sfarsit";
   }
```

Programul *catch_3.cpp* utilizează o serie de instrucţiuni *if* în cadrul blocului *try* pentru a lansa diferite excepţii în trei apeluri diferite de funcţii. Atunci când compilaţi şi executaţi programul *catch_3.cpp*, ecranul dumneavoastră va afişa următorul rezultat:

```
Start
Am captat exceptia #: 0
Am captat exceptia de tip sir de caractere: Sir de caractere
Am captat exceptia #: 123.23
Sfarsit
C:\>
```

1135 UTILIZAREA OPERATORULUI PUNCTE DE SUSPENSIE (...) CU EXCEPŢII



După cum aţi învăţat, în secţiunile precedente, programele dumneavoastră pot capta excepţii din cadrul mai multor blocuri *try* sau să utilizeze mai multe instrucţiuni *catch* într-un singur bloc *try*. Însă, programele dumneavoastră pot, de asemenea, să utilizeze operatorul *puncte de suspensie* pentru a capta orice tip de erori care apar într-un singur bloc *try*. Pentru a capta toate erorile dintr-un bloc *try*, veţi construi blocul *try* în forma generalizată prezentată mai jos:

```
try
  {
    // instructiuni
  }
catch(...)
  {
    // tratarea exceptiei
  }
```

CAPTAREA TUTUROR EXCEPȚIILOR DINTR-UN SINGUR BLOC TRY

C/C++1136

După cum ați învățat în secțiunea 1135, programele dumneavoastră pot utiliza operatorul *puncte de suspensie* pentru a capta excepții multiple. După cum veți învăța în secțiunea 1137, puteți utiliza operatorul *puncte de suspensie* împreună cu un bloc *try-catch* standard. Însă, puteți să utilizați, de asemenea, operatorul *puncte de suspensie* de sine stătător, pentru a capta mai multe excepții de tipuri necunoscute sau diferite. Următorul program, *catch_mlt.cpp*, utilizează operatorul *puncte de suspensie* pentru a capta trei excepții de trei tipuri diferite:

```
#include <iostream.h>

void XHandler(int test)
  {
    try
      {
        if(test==0)
          throw test;
        if(test==1)
          throw 'a';
        if(test==2)
          throw 123.23;
      }
    catch(...)
      {
        cout << "Am captat una." << endl;
      }
  }

void main(void)
  {
    cout << "Start: " << endl;
    XHandler(0);
    XHandler(1);
    XHandler(2);
    cout << "Sfarsit";
  }
```

Notați că, spre deosebire de programul *catch_3.cpp* din secțiunea 1134, programul *catch_mlt.cpp* utilizează doar o singură instrucțiune *catch* pentru a capta toate cele trei erori. Atunci când compilați și executați programul *catch_mlt.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Start:
Am captat una.
Am captat una.
Am captat una.
Sfarsit
C:\>
```

Trebuie să observați că, în programul *catch_mlt.cpp*, blocul dumneavoastră *catch* care utilizează operatorul *puncte de suspensie* nu poate distinge tipul de excepție pe care îl lansează programul, ceea ce înseamnă că prelucrările din cadrul blocului *catch* trebuie să fie independente de tipul erorii. Secțiunea 1137 explică modul în care să includeți într-un singur bloc tratarea excepțiilor atât dependentă, cât și independentă de tipul erorilor.

1137 CAPTAREA EXCEPȚIILOR EXPLICITE ȘI A EXCEPȚIILOR GENERICE ÎNTR-UN SINGUR BLOC TRY

C/C++

După cum ați învățat, puteți utiliza operatorul *puncte de suspensie* pentru a capta mai multe excepții de tipuri necunoscute. Mult mai des, însă, veți dori ca programele dumneavoastră să capteze excepții explicite și să răspundă excepțiilor în moduri specifice. Dacă tratați excepții specifice, explicite, într-un bloc *try* dat, puteți să utilizați în plus operatorul puncte de suspensie pentru a capta toate excepțiile care nu sunt de tipurile așteptate. Deoarece limbajul C++ vă permite să captați mai multe tipuri de excepții într-un bloc *try* dat, puteți cu ușurință să creați secvențe de excepții care vă permit să captați atât excepții explicite, cât și generice și să le manevreze în mod diferit. Următorul program, *exp_neexp.cpp*, capteaza atât excepții de tip întreg, cât și necunoscute, într-un singur bloc *try*:

```
#include <iostream.h>

void XHandler(int test)
  {
    try
      {
        if(test==0)
          throw test;
        if(test==1)
          throw 'a';
        if(test==2)
          throw 123.23;
      }
    catch(int i)
      {
        cout << "Am captat o exceptie de tip intreg." << endl;
      }
    catch(...)
```

```
      {
        cout << "Am captat una." << endl;
      }
  }

void main(void)
  {
    cout << "Start: " << endl;
    XHandler(0);
    XHandler(1);
    XHandler(2);
    cout << "Sfarsit";
  }
```

Programul *exp_neexp.cpp* va capta excepția de tip întreg lansată către blocul *catch* explicit și excepțiile de tip *double* și *char* lansate către blocul *catch* generic. Atunci când compilați și executați programul *exp_neexp.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Start
Am captat o exceptie de tip intreg.
Am captat una.
Am captat una.
Sfarsit
C:\>
```

RESTRICȚIONAREA EXCEPȚIILOR

C/C++1138

Pe măsură ce programele dumneavoastră devin mai complexe, ele vor apela frecvent funcții din cadrul unui bloc *try*. Atunci când programele dumneavoastră apelează funcții dintr-un bloc *try*, puteți restricționa tipurile de excepții pe care funcția apelată le poate lansa. De asemenea, puteți preveni ca funcția să lanseze vreun tip de excepții. Pentru a restricționa excepțiile pe care funcțiile dumneavoastră le pot lansa, trebuie să adăugați o clauză *throw* definiției funcției. Forma generală a funcției cu restricții de lansare este arătată mai jos:

```
tip-retur nume-functie(lista-argumente) throw(lista-tipuri)
  {
    // codul functiei
  }
```

Atunci când declarați o funcție cu clauza *throw*, ea poate să lanseze doar acele tipuri pe care le detaliați în cadrul listei *lista-tipuri*. Dacă funcția lansează orice alt tip de excepție, programul se va termina în mod anormal (așa cum ați văzut în secțiunea 1130, cu programul *catch_d.cpp*). Dacă nu doriți ca o funcție să lanseze vreo excepție, utilizați o listă *lista-tipuri* vidă. Pentru a înțelege mai bine modul în care programele dumneavoastră pot limita excepțiile lansate de o funcție, analizați următorul program, *fun_thrw.cpp*, care limitează funcția *XHandler* la lansarea excepțiilor de tip *int*, *char* și *double*.

```
#include <iostream.h>

void XHandler(int test) throw(int, char, double)
  {
    if(test==0)
```

```
      throw test;
    if(test==1)
      throw 'a';
    if(test==2)
      throw 123.23;
}
void main(void)
  {
    cout << "Start: " << endl;
    try {
      XHandler(0);    // incearca sa transmita 1 si 2 pentru
                      // raspunsuri diferite
    }
    catch(int i) {
      cout << "Am captat un intreg." << endl;
    }
    catch(char c) {
      cout << "Am captat un caracter." << endl;
    }
    catch(double d) {
      cout << "Am captat un double." << endl;
    }
    cout << "Sfarsit";
  }
```

Este important să înţelegeţi că utilizarea clauzei *throw* în cadrul declaraţiei unei funcţii, limitează numai excepţiile pe care funcţia le poate lansa înapoi, către blocul *try* din care programul a apelat funcţia. În cadrul funcţiei, puteţi să utilizaţi încă un bloc *try* pentru a capta orice excepţie, chiar şi excepţiile care nu sunt listate în cadrul listei *lista-tipuri*. Atunci când compilaţi şi executaţi programul *fun_thrw.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Start
Am captat un intreg.
Sfarsit
C:\>
```

Următoarea declaraţie a funcţiei *XHandler* va împiedica funcţia de la lansarea oricărei excepţii către blocul apelant:

```
void XHandler(int test) throw()
  {
    if(test==0)
      throw test;
    if(test==1)
      throw 'a';
    if(test==2)
      throw 123.23;
  }
```

Deoarece instrucțiunea *throw* din declarația funcției nu include nici o valoare în lista *lista-tipuri*, funcția nu va lansa nici o excepție. Dacă compilați și executați fișierul *fun_ntbr.cpp*, pe ecranul dumneavoastră vor apărea următoarele:

```
Start:
Abnormal program termination
C:\>
```

RELANSAREA UNEI EXCEPȚII

C/C++1139

Pe măsură ce programele dumneavoastră de tratare a excepțiilor devin mai complexe, poate fi necesar să se relanseze o excepție din interiorul unui handler de excepții. Dacă relansați o excepție, limbajul C++ va transmite excepția unui bloc *try* exterior. Cea mai probabilă situație în care veți utiliza astfel de prelucrări în cadrul programelor dumneavoastră va fi atunci când veți dori să tratați o excepție în cadrul a două programe handler distincte. De exemplu, un handler din cadrul unei funcții poate trata un aspect al excepției, iar un handler din afara funcției, poate trata un alt aspect. Este important să înțelegeți că relansarea unei excepții va transmite *imediat* excepția către un handler exterior (cel interior nu va trata excepția relansată). Pentru a înțelege mai bine prelucrările pe care limbajul C++ le efectuează atunci când relansați o excepție, analizați următorul program, *doua_exp.cpp*, care lansează o excepție în interiorul unei funcții, relansează excepția și captează din nou excepția, în afara funcției:

```
#include <iostream.h>
void XHandler(void)
  {
    try {
      throw "salut";
    }
    catch(char *) {
      cout << "Am captat char * in XHandler." << endl;
      throw;
    }
  }
void main(void)
  {
    cout << "Start: " << endl;
    try
      {
        XHandler();
      }
    catch(char *)
      {
        cout << "Am captat char * in main." << endl;
      }
    cout << "Sfarsit";
  }
```

Atunci când compilați și executați programul *doua_exp.cpp*, el va apela funcția *XHandler*. Blocul *try* intern va capta instrucțiunea *throw* din funcția *XHandler* și va genera un mesaj. Blocul *try* intern va relansa excepția, iar blocul *try* din *main* va recapta apoi excepția. Atunci când executați programul *doua_exp.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Start:
Am captat char * in XHandler.
Am captat char * in main.
Sfarsit
C:\>
```

1140 APLICAȚII ALE TRATĂRII EXCEPȚIILOR C/C++

După cum ați învățat, C++ dispune de capacități de tratare a excepțiilor prin care vă furnizează o structură pe care o puteți utiliza pentru a face ca programele dumneavoastră să răspundă la evenimente anormale sau neașteptate. Deoarece veți utiliza *try* pentru controlul erorilor, veți prelucra, în mod obișnuit, aceste erori din cadrul programelor dumneavoastră într-o manieră utilă pentru programe (spre deosebire de simplele mesaje de eroare pe care le-ați generat până acum în blocul *catch*). Pentru a înțelege mai bine tratarea excepțiilor utilizând *try*, analizați următorul program, *catch_dz.cpp*, care generează și captează o eroare de împărțire prin zero:

```
#include <iostream.h>

void divide(double a, double b)
  {
    try
     {
      if(!b)
        throw b; // verifica daca divide la zero
      cout << "Rezultat: " << a/b << endl;
     }
    catch(double b)
     {
      cout << "Nu se poate divide la zero." << endl;
     }
  }

void main(void)
  {
    double i, j;
    do {
      cout << "Introduceti numaratorul (0 pentru stop):" << endl;
      cin >> i;
      cout << "Introduceti numitorul: " << endl;
      cin >> j;
      divide (i,j);
    }
    while (i !=0);
  }
```

Comparați blocul *try* din această secțiune cu programul handler pentru erori matematice pe care l-ați proiectat în secțiunea 351. Blocul *try* este mai ușor de implementat, mai simplu de înțeles și mai eficient decât acel handler de erori matematice proiectat de dumneavoastră anterior. Datorită faptului că tratarea excepțiilor pune la dispoziție mai multă putere și este mai ușor de folosit, pe măsură ce programele dumneavoastră devin mai complexe, veți descoperi că o veți utiliza frecvent pentru protecția împotriva evenimentelor neașteptate, în special introduse de utilizator.

UTILIZAREA ARGUMENTELOR IMPLICITE ALE FUNCȚIILOR

C/C++1141

După cum ați învățat, C++ vă permite crearea de funcții cu valori implicite ca parametri. Funcțiile dumneavoastră vor utiliza numai valorile implicite pentru parametri, atunci când instrucțiunea de apelare nu conține explicit valori pentru parametri sau când programul nu a inițializat încă variabilele pe care instrucțiunea de apelare le utilizează ca parametri. Valorile implicite sunt deseori utile pentru programele dumneavoastră, pentru că ele pot reduce sau elimina necesitatea unor programe handler pentru excepții.

Veți utiliza, însă, în general, valorile parametru implicite în cadrul funcțiilor dumneavoastră atunci când programul apelează în mod repetat funcția cu aceeași valoare și numai din când în când cu o valoare diferită. De exemplu, următorul program, *clrscr.cpp*, utilizează caracterul de linie nouă pentru a șterge ecranul (după cum ați învățat, utilizarea caracterului de linie nouă nu este cea mai eficientă modalitate de ștergere a ecranului, dar este o tehnică utilă pentru acest exemplu). Pentru că multe ecrane au înălțimea de douăzeci și cinci de linii, funcția *clrscr* este poziționată implicit la 25 de linii:

```
#include <iostream.h>

void clrscr(int dimens=25)
  {
    while(dimens > 0)
      {
        cout << endl;
        dimens--;
      }
  }
void main(void)
  {
    int i;
    for(i=0; i<30; i++)
      cout << i << endl;
    cin.get();
    clrscr(); // sterge 25 de linii
    for(i=0; i<30; i++)
      cout << i << endl;
    cin.get();
    clrscr(10); // sterge 10 linii
  }
```

1142 EVITAREA ERORILOR CU ARGUMENTELE IMPLICITE ALE FUNCȚIILOR

C/C++

După cum ați învățat, puteți utiliza frecvent argumente implicite în cadrul programelor dumneavoastră. Mulți programatori utilizează argumentele implicite chiar cu funcțiile constructor, cum arătăm mai jos:

```
class cub {
   int x, y, z;
  public:
   cub(int i = 0, int j = 0, int k = 0)
    {
      x = i;
      y = j;
      z = k;
    }
    // restul definitiei
  };
```

Utilizarea valorilor implicite în cadrul funcțiilor constructor poate contribui la evitarea supraîncărcării funcților constructor. Însă, ca și în cazul multor tehnologii de simplificare pe care le-ați învățat (cum ar fi funcțiile generice), ar trebui să acordați atenție limitării numărului de parametri pentru care stabiliți valori implicite. Ca regulă generală, nu trebuie să declarați o valoare implicită pentru un parametru decât dacă funcția utilizează valoarea implicită în 75% din cazuri sau mai mult. Dacă funcția utilizează rar valoarea implicită, specificarea valorii implicite nu este utilă, ci deseori este distructivă pentru programele dumneavoastră și echivocă pentru alți programatori care citesc codul. Utilizați valorile implicite în cadrul programelor dumneavoastră atât cât este necesar, dar aveți grijă să nu abuzați de ele.

1143 ARGUMENTELE IMPLICITE ȘI SUPRAÎNCĂRCAREA FUNCȚIILOR

C/C++

După cum ați învățat, programele dumneavoastră pot utiliza argumente implicite pentru a simplifica procesul de tratare a parametrilor funcțiilor. După cum sugerează secțiunile precedente, puteți utiliza argumente implicite pentru a preveni supraîncărcarea unor funcții, cum ar fi funcțiile constructor. De asemenea, puteți utiliza argumente implicite ca o metodă rapidă și eficientă de punere la dispoziție a unor funcții supraîncărcate. Să presupunem că doriți să creați două versiuni ale propriei dumneavoastră funcții *concatsir*, care concatenează două șiruri de caractere. Prima versiune lucrează în exact aceeași manieră cu *strcat*, concatenând întregul conținut al unui șir la capătul celuilalt. Cea de a doua versiune acceptă un al treilea parametru care specifică numărul de caractere pe care funcția trebuie să le concateneze (similar cu *strncat*). Pentru a supraîncărca funcția, veți declara antetele pentru funcția *concatsir*, ca mai jos:

```
void concatsir(char *s1, char *s2);
void concatsir(char *s1, char *s2, int lung);
```

Ca alternativă, puteți utiliza o valoare implicită pentru parametrul *lung*, pe care funcția o va testa înainte să își înceapă prelucrarea, ca mai jos:

```
void concatsir(char *s1, char *s2, int lung = 0)
  {
    if(lung = 0)
      {
        // prelucrare
      }
    else
      {
        // alte prelucrari
      }
  }
```

În exemplul precedent, utilizarea unui parametru implicit face programul mai ușor de înțeles. Ca regulă, însă, încercați să utilizați acea formă de funcție care este cea mai semnificativă și care efectuează prelucrările în cel mai evident mod – atât pentru dumneavoastră cât și pentru alți programatori care vă citesc codul.

CREAREA FUNCȚIILOR DE CONVERSIE

C/C++1144

Pe măsură ce programele dumneavoastră devin mai complexe, puteți să utilizați un obiect al unei clase într-o expresie care implică alte tipuri de date. După cum ați învățat, puteți să utilizați frecvent funcții operator supraîncărcate, pentru a vă ajuta în asemenea cazuri. Însă, în alte cazuri, probabil că doriți pur și simplu o conversie de tip, de la tipul clasei la tipul destinației. Pentru a vă ajuta în privința conversiilor de tip, limbajul C++ vă premite să creați *funcții de conversie* personalizate. O funcție de conversie convertește clasa dumneavoastră într-un tip compatibil cu restul unei expresii. Forma generală a unei funcții de conversie de tip este arătată mai jos:

```
operator tip(void) { return valoare; }
```

În formatul generalizat, *tip* corespunde tipului destinație, iar *valoare* este valoarea obiectului după conversie. Funcțiile de conversie nu acceptă parametri, doar returnează o valoare de tipul *tip*. Pentru a înțelege mai bine modul de lucru al funcțiilor de conversie, studiați următorul exemplu, care convertește valori de tipul *stiva* la valori de tipul *int*, returnând valoarea *vfs* (vârful stivei) către expresie. Programul *conv_stv.cpp* utilizează clasa stivă pe care ați creat-o în secțiunile precedente:

```
#include <iostream.h>
const int DIM=100;
class stiva {
    int stv[DIM];
    int vfs;
  public:
    stiva(void) {vfs=0;}
    void depune(int i);
    int extrage(void);
```

```
    operator int(void) {return vfs;} // converteste stiva in int
  };
void stiva::depune(int i)
  {
    if(vfs==DIM)
      {
        cout << "Stiva este plina." << endl;
        return;
      }
    stv[vfs++] = i;
  }
int stiva::extrage(void)
  {
    if(vfs==0)
      {
        cout << "Stiva este depasita in jos." << endl;
        return 0;
      }
    return stv[--vfs];
  }
void main(void)
  {
    stiva stv;
    int i, j;
    for(i=0; i<20; i++)
      stv.depune(i);
    j = stv; // converteste in int
    cout << j << " elemente in stiva." << endl;
    cout << (DIM - stv) << " spatii libere." << endl;
  }
```

1145 UTILIZAREA FUNCȚIILOR CONVERSIE PENTRU A PERFECȚIONA PORTABILITATEA TIPURILOR



După cum ați învățat în secțiunea 1144, programele dumneavoastră pot utiliza funcțiile de conversie pentru a converti obiecte ale unei clase la un alt tip. Una dintre cele mai bune utilizări ale funcțiilor de conversie este pentru a crește portabilitatea tipurilor obiectelor și prin aceasta ele să devină mai utile. De exemplu, următorul program, *ptr_dbl.cpp*, utilizează clasa *putere*. Însă, programul *ptr_dbl.cpp* convertește, de asemenea, *putere* într-un tip *double* în cadrul expresiilor, ceea ce permite utilizarea rezultatului operației *putere* în cadrul altor ecuații matematice, cum arătăm mai jos:

```
#include <iostream.h>
class putere {
```

```
  double b;
  int e;
  double val;
public:
  putere(double baza, int exp);
  putere operator+(putere ob)
    {
      double baza;
      int exp;
      baza = b + ob.b;
      exp = e + ob.e;
      putere temp(baza, exp);
      return temp;
    }
    operator double(void) {return val;} // converteste la double
  };

putere::putere(double baza, int exp)
  {
    b = baza;
    e = exp;
    val = 1;
    if (exp!=0)
      while(exp-- > 0)
        val *= b;
  }

void main(void)
  {
    putere puterel(4.0, 2);
    double doubl;
    doubl = puterel; // converteste la double
    cout << (doubl + 100.2) << endl;
    putere putere2(3.3, 3), putere3(0,0);
    putere3 = puterel + putere2; // nu converteste
    doubl = putere3; // converteste la double
    cout << doubl;
  }
```

Programul *ptr_dbl.cpp* declară variabila *putere1* (de tip *putere*) și variabila *doub1*, de tip *double*. Apoi el convertește *putere1* la o valoare *double* și afișează valoarea convertită. După aceea, programul declară două noi variabile de tip *putere*, *putere2* și *putere3*, asupra cărora efectuează prelucrarea și apoi afișează. Atunci când compilați și executați programul *ptr_dbl.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
116.2
20730.7
C:\>
```

1146 FUNCȚIILE DE CONVERSIE ȘI OPERATORII SUPRAÎNCĂRCAȚI

C/C++

După cum ați învățat, ați putea efectua în principiu aceeași prelucrare pe care ați efectuat-o în secțiunile 1144 și 1145 prin supraîncărcarea operatorilor corespunzători claselor *stiva* și respectiv, *putere*. Din păcate, așa cum ați învățat, trebuie să supraîncărcați operatorii în mod diferit pentru a efectua fiecare dintre următoarele atribuiri:

```
x = putere + 102.65;
x = 102.65 + putere;
```

Însă, în locul utilizării operatorilor supraîncărcați, puteți crea o funcție de conversie, cum detaliază secțiunea 1145. O funcție de conversie vă ajută să evitați supraîncărcarea mai multor operatori, utilizând funcții *friend* sau executând alte prelucrări complexe, repetitive, pentru a converti un obiect la o altă valoare. Pe de altă parte, dacă lucrați cu o clasă și doriți să adăugați *102.65* fiecărui membru al acelei clase, funcția de conversie nu vă ajută. La fel cum ați observat și în cazul parametrilor impliciți și al funcțiilor supraîncărcate, decizia de utilizare a unei funcții de conversie în locul unui operator supraîncărcat va diferi de la un program la altul și de la o clasă la alta. Ca și în exemplele anterioare, ar trebui să luați decizia pe baza aplicației specifice a clasei în cadrul programului dumneavoastră.

1147 NOII OPERATORI C++ DE MODELARE

C/C++

După cum ați învățat în secțiunile precedente, C acceptă un operator de modelare, pe care programele dumneavoastră îl pot utiliza pentru a modela o valoare într-un tip destinație. C++ definește patru noi operatori de modelare, listați în tabelul 1147

Nume	Forma generală
const_cast	const_cast<tip>(obiect)
dynamic_cast	dynamic_cast<tip>(obiect)
reinterpret_cast	reinterpret_cast<tip>(obiect)
static_cast	static_cast<tip>(obiect)

Tabelul 1147 *Noii operatori de modelare acceptați de C++.*

Programele dumneavoastră pot continua să folosească operatorii C de modelare și ar putea utiliza și noii operatori de modelare din C++, în concordanță cu prelucrările programului. Veți afla mai multe despre fiecare operator nou de modelare în următoarele secțiuni.

1148 UTILIZAREA OPERATORULUI CONST_CAST

C/C++

După cum ați învățat, C++ definește câțiva noi operatori de modelare pe care îi puteți utiliza în cadrul programelor dumneavoastră. În cadrul programelor dumneavoastră, veți utiliza operatorul *const_cast* pentru a suprapune în mod explicit o declarație anterioară *const* sau *volatile*. Tipul de destinație al modelării trebuie să fie același cu tipul sursă, cu excepția modificării atributelor *const* sau *volatile*. Veți utiliza mult mai frecvent *const_cast* pentru a elimina un atribut constant al unei valori – cu alte cuvinte, pentru a face modificabilă o valoare pe care anterior ați definit-o drept constantă.

Programele dumneavoastră pot utiliza operatorul *const_cast* pentru a converti în mod explicit un pointer către orice tip de obiect sau un pointer către date membre la un tip care este identic, cu excepţia atributelor *const*, *volatile* şi *_unaligned*. Pentru pointeri şi referinţe, rezultatul se referă la obiectul iniţial. Pentru pointeri la membri, rezultatul se referă la acelaşi membru ca pointerul iniţial (nemodelat) la respectivul membru. De exemplu, următorul program, *const.cpp*, nu se va compila, datorită faptului că nu puteţi atribui o valoare constantă a pointerului la o valoare normală a pointerului:

```
#include <stdio.h>

class c {
  public:
    int j;
    c(void) {j=10;}
  };

void Imp(const c* Obiect)
{
    c* Nou = Obiect;
    Nou->j +=5;
    printf("%d\n", Nou->j);
}

void main(void)
{
    const c Exemplu;
    Imp(&Exemplu);
}
```

Dacă modificaţi funcţia *Imp*, însă, pentru a utiliza operatorul *const_cast*, programul se va compila şi se va executa corect, aşa cum arătăm în continuare, în fragmentul de cod din programul *const_cast.cpp*:

```
void Imp(const c* Obiect)
{
   c* Nou = const_cast<c*> (Obiect);
   Nou->j +=5;
   printf("%d\n", Nou->j);
}
```

Modelarea vă permite să creaţi pointerul *Nou* ca un pointer modificabil. Însă, trebuie să folosiţi cu grijă operatoul *const_cast*. În funcţie de tipul obiectului referenţiat, o operaţie de scriere prin pointerul, referinţa sau pointerul la un membru dată ce rezultă, pot duce la o evoluţie necunoscută şi neanticipată. Următorul fragment de cod va duce la o eroare:

```
class X {};
class Y : public X {};

const X x;
Y y = const_cast<Y> (x); // eroare
```

Codul provoacă o eroare deoarece nu puteți utiliza operatorul *const_cast* pentru a face conversia din *X* în *Y*. Puteți să utilizați operatorul *const_cast* doar pentru a elimina modificatorii *const*, *volatile* și *_unaligned*. Dacă doriți să convertiți obiectul constant *x* la obiectul ne-constant *y*, trebuie să construiți operația de modelare ca mai jos:

```
Y y = (const_cast<Y>)(static_cast<const Y>(x));
```

Observație: *Operatorul* ***const_cast*** *convertește valoarea pointer NULL la valoarea pointer NULL a tipului de destinație.*

1149 UTILIZAREA OPERATORULUI DYNAMIC_CAST C/C++

După cum ați învățat, C++ dispune de câțiva noi operatori de modelare, pe care îi puteți utiliza în cadrul programelor dumneavoastră. Operația *dynamic_cast* efectuează o modelare în timpul rulării și verifică validitatea modelării. Dacă programul nu poate efectua modelarea, modelarea va eșua și expresia va fi evaluată la NULL. În general, veți utiliza operatorul *dynamic_cast* pentru a efectua modelări asupra obiectelor de tip polimorfic. De exemplu, *dynamic_cast* poate returna un pointer la un obiect derivat dat fiind un pointer la clasa de bază polimorfică. Pentru a înțelege mai bine prelucrările pe care le efectuează operatorul *dynamic_cast*, analizați următorul exemplu de program, *dyn_cast.cpp*, care generează o clasă din două clase de bază și încearcă să modeleze pointerii din clasele de bază la clasa derivată:

```
#include <iostream.h>
#include <typeinfo.h>

class Baza1
  {
    virtual void f(void) { /* O functie virtuala face clasa
      polimorfica */ }
  };

class Baza2 { };
class Derivata : public Baza1, public Baza2 { };

void main(void)
  {
    try
      {
        Derivata d, *pd;
        Baza1 *b1 = &d;

        // Efectueaza modelarea de la Baza1 la Derivata.
        if ((pd = dynamic_cast<Derivata *>(b1)) != 0)
          {
            cout << "Pointerul rezultat este de tip "
               << typeid(pd).nume() << endl;
          }
        else
          throw Gresit_mod();
```

```
        // Modeleaza de la prima clasa de baza la ultima clasa
        // derivata si inapoi la alta clasa baza accesibila.
        Baza2 *b2;
        if ((b2 = dynamic_cast<Baza2 *>(b1)) != 0)
          {
            cout << "Pointerul rezultat este de tip "
              << typeid(b2).name() << endl;
          }
        else
          throw Gresit_mod();
      }
    catch (Gresit_mod)
      {
        cout << "dynamic_cast esuat" << endl;
        exit(1);
      }
    catch (...)
      {
        cout << "Eroare de tratare a erorii." << endl;
        exit(1);
      }
  }
```

Atunci când executați programul *dyn_cast.cpp*, acesta va efectua două încercări de modelare. Mai întâi, programul va încerca să modeleze de la un pointer din *Bază1* la un pointer din *Derivata*. Apoi, el va modela de la prima clasă de bază la ultima clasă derivată și ulterior va încerca să modeleze parcurgând arborele invers, la altă clasă de bază. Atunci când compilați și executați programul *dyn_cast.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Pointerul rezultat este de tip Derivata *
Pointerul rezultat este de tip Baza1 *
C:\>
```

Observație: *Trebuie să compilați programul* ***dyn_cast.cpp*** *cu opțiunea de compilare Generate RTTI activată, altfel programul nu va putea efectua identificarea tipului în timpul rulării.*

UTILIZAREA OPERATORULUI REINTERPRET_CAST

C/C++1150

După cum ați învățat, C++ dispune de câțiva operatori de modelare noi, pe care îi puteți folosi în cadrul programelor dumneavoastră. Operatorul *reinterpret_cast* modelează un tip la un alt tip fundamental diferit, incompatibil. De exemplu, operatorul *reinterpret_cast* ar putea modela un pointer de tip *putere* la un pointer de tip *int*. Desigur, operatorul *reinterpret_cast* constituie o sursă de confuzii în cadrul programelor dumneavoastră și ar trebui să-l evitați când utilizarea lui nu este absolut necesară. Următorul program, *ren_cast.cpp*, utilizează operatorul *reinterpret_cast* pentru a converti un pointer *char* la un pointer *int*:

```
#include <iostream.h>
void main(void)
  {
    int i;
    char *p = "Acesta este un sir de caractere.";
    i = reinterpret_cast<int> (p); // modeleaza pointerul char
                                   // la intreg
    cout << i;
  }
```

Atunci când compilați și executați programul *ren_cast.cpp*, acesta va afișa o ieșire lipsită de sens, deoarece modelarea de la pointerul *char* la un pointer *int* determină pointerul *int* să returneze un rezultat bizar. Dacă rulați programul într-un sistem pe 16 biți, *int* va returna *char('A')*. Dacă, însă, rulați programul într-un un sistem pe 32 de biți care acceptă întregi pe 32 de biți, pointerul *i* returnează echivalentul pe biți al șirului constant 'Ac'. Într-un sistem pe 32 de biți, ieșirea va fi asemănătoare cu următoarea:

```
4247824
C:\>
```

1151 UTILIZAREA OPERATORULUI STATIC_CAST C/C++

După cum ați învățat, C++ dispune de câțiva operatori de modelare noi pe care îi puteți utiliza în cadrul programelor dumneavoastră. Programele dumneavoastră vor utiliza operatorul *static_cast* pentru a efectua o modelare nepolimorfică. Cu alte cuvinte, puteți utiliza operatorul *static_cast* pentru a modela un pointer al clasei de bază la un pointer al clasei derivate. Veți utiliza operatorul *static_cast* în cadrul programelor dumneavoastră așa cum arată următorul prototip:

```
static_cast< T > (argument)
```

În forma generalizată prezentată mai sus, *T* este un pointer, referință, tip aritmetic sau tip enumerat. Tipul pentru *argument* trebuie să fie același cu tipul lui *T*. Compilatorul trebuie să cunoască atât *T*, cât și *argument* la momentul compilării. Dacă programul dumneavoastră poate converti un tip de bază la un alt tip printr-o metodă de conversie de care dispune deja limbajul, efectuarea unei astfel de conversii utilizând în schimb operatorul *static_cast* va realiza aceeași conversie. În plus, programele dumneavoastră pot utiliza operatorul *static_cast* pentru a converti tipurile întregi la tipuri enumerare. O solicitare de conversie a parametrului *argument* la o valoare care nu este un element al *enumerării* returnează o valoare *undefined*. Pointerul *NULL* se convertește la el însuși (cu alte cuvinte, la modelarea cu operatorul *static_cast* a lui *NULL*, rezultă *NULL*) și de aceea, programele dumneavoastră nu trebuie să utilizeze *static_cast* cu pointerul *NULL*.

Programul dumneavoastră poate converti un pointer la tipul unui obiect la un pointer la tipul altui obiect. Observați că chiar și indicarea unor tipuri similare poate provoca probleme de accesare dacă nu aliniați corespunzător tipurile similare. Puteți converti explicit un pointer la o clasă *baza* la un pointer la altă clasă *derivata*, dacă clasa *baza* este o clasă de bază pentru *derivata*. Programul poate să facă o conversie statică numai în următoarele condiții:

- Dacă există o conversie neambiguă de la *derivata* la *baza*.
- Dacă *baza* nu este o clasă de bază virtuală.

Operatorul *static_cast* poate converti explicit un obiect la tipul referință *baza&*, atunci când compilatorul poate converti explicit un pointer la acel obiect la pointerul tipului *baza**. Rezultatul conversiei *static_cast* este *lvalue*. Programul nu va apela funcții constructor sau de conversie ca rezultat al unei modelări la o referință. În schimb, programul poate converti un obiect sau o valoare la un obiect clasă numai dacă ați declarat constructorul sau operatorul de conversie corespunzători pentru acea modelare. Puteți converti explicit un pointer la un membru într-un alt tip de pointer la membru, numai dacă ambele tipuri sunt pointeri către membri ai aceleiași clase sau pointeri către membri din două clase. Dacă atât argumentul cât și cazul rezultat sunt pointeri către membri din două clase, programul trebuie să deriveze clasa referențiată în mod direct dintr-un pointer din cealaltă clasă.

Atunci când *T* este o referință, rezultatul operației *static_cast* este o *lvalue*. Rezultatul unei modelări a unui pointer sau referințe se referă la expresia inițială.

NAMESPACE

C/C++1152

Pe măsură ce programele dumneavoastră devin mai complexe, aplicațiile dumneavoastră se vor baza, în cele din urmă, pe mai multe fișiere sursă. În plus, mai mulți programatori pot crea și întreține fiecare fișier sursă. Până la urmă, veți organiza și lega fișierele separate pentru a produce aplicația finală. De obicei, organizarea fișierelor cere ca toate numele pe care un fișier sursă nu le încapsulează în cadrul unui *nume de spațiu* (*namespace*) definit (cu alte cuvinte, care nu are o limită a domeniului de valabilitate, cum ar fi corpul funcției, corpul clasei sau unitatea de conversie) trebuie să partajeze același spațiu de nume global. De aceea, pentru multe dintre definițiile de nume pe care compilatorul le descoperă în timpul editării legăturilor între module separate este necesară o modalitate de a distinge fiecare nume. Cuvântul cheie *namespace* din C++ oferă soluția problemei „impactului numelui" cu domeniul global de valabilitate.

Cuvântul cheie *namespace* vă permite partiționarea unei aplicații în mai multe subsisteme. Fiecare subsistem poate defini și opera în cadrul propriului lui domeniu de valabilitate (scape). Fiecare programator poate introduce identificatori convenabili într-un subsistem, fără să se teamă că alți programatori vor utiliza acești identificatori în cadrul propriului lor subsistem. Fiecare nume de spațiu utilizează un identificator unic. Atunci când programele dumneavoastră definesc numele de spații, compilatorul cunoaște domeniul de valabilitate al subsistemului pe parcursul întregii aplicații prin fiecare identificator unic de nume de spațiu.

Utilizarea numelor de spații ale limbajului C++ presupune două etape. Prima este utilizarea cuvântului cheie *namespace* pentru a identifica numele de spațiu. A doua etapă este invocarea cuvântului cheie *using* pentru a accesa elementele unui nume de spațiu anterior identificat.

UTILIZAREA CUVÂNTULUI CHEIE NAMESPACE

C/C++1153

După cum ați învățat în secțiunea 1152, aspectele legate de domeniul de valabilitate și denumiri devin mai importante pe măsură ce programele dumneavoastră devin mai complexe. Pentru a vă ajuta să preveniți conflictele între variabile și alte nume, C++ vă pune la dispoziție cuvântul cheie *namespace*. Veți utiliza cuvântul cheie *namespace* în cadrul programelor dumneavoastră asemănător cu o definiție de structură, de tip enumerat, de uniune

sau de clasă, aşa cum arătăm mai jos în forma generalizată de implementare a cuvântului cheie *namespace*:

```
namespace nume {
    // declaratii de obiecte
}
```

În cadrul propriilor dumneavoastră programe, puteţi include în cadrul definiţiei *namespace* variabile de orice tip pe care compilatorul îl cunoaşte deja (fie tipuri simple C şi C++ sau clase definite anterior, structuri sau uniuni). Puteţi, de asemenea, să declaraţi funcţii *inline* în cadrul unui *namespace*. De exemplu, următorul fragment de cod declară două variabile şi o funcţie în cadrul numelui de spaţiu *limitat*:

```
namespace limitat {
    int i, k;
    void exemplu(int j) { cout << j << endl; }
}
```

În fragmentul de cod precedent, *i*, *k* şi funcţia *exemplu* sunt părţi ale spaţiului *limitat*. Deoarece un nume de spaţiu defineşte un domeniu de valabilitate, trebuie să utilizaţi operatorul de rezoluţie a domeniului pentru a face referire la obiectele definite în interiorul numelui de spaţiu. De exemplu, pentru a atribui valoarea 10 variabilei *k*, trebuie să utilizaţi o instrucţiune asemănătoare cu următoarea:

```
limitat:: k = 10;
```

1154 UTILIZAREA INSTRUCŢIUNII USING CU NAMESPACE

C/C++

După cum aţi învăţat în secţiunea 1153, atunci când programele dumneavoastră utilizează *nume de spaţii*, trebuie să rezolvaţi referinţa la obiectele din cadrul unui nume de spaţiu cu operatorul de rezoluţie a domeniului de valabilitate (scope). Însă, dacă programele dumneavoastră vor utiliza frecvent membrii unui spaţiu, puteţi utiliza instrucţiunea *using* pentru a simplifica accesul programului la aceşti membri. Instrucţiunea *using* are două forme generale, ca mai jos:

```
using namespace nume;
using nume::membru;
```

Prima formă vă permite accesul la întregul spaţiu. În a doua formă, definiţi numai membrii specificaţi ai spaţiului pe care doriţi să-i accesaţi. În esenţă, prima formă face întregul spaţiu public, iar a doua formă vă permite să încapsulaţi anumiţi membri în interiorul spaţiului. Pentru a înţelege mai bine cele două forme ale instrucţiunii *using*, analizaţi următorul fragment de cod, care utilizează ambele forme:

```
using limitat::k; // face vizibil doar membrul k
k = 10;
using namespace limitat; // face vizibil intregul spatiu limitat
k = 10;
```

IDENTIFICAREA TIPULUI ÎN TIMPUL RULĂRII

C/C++1155

O importantă completare pe care o aduc noile compilatoare de C++ este identificarea tipului în timpul rulării (*run-time type identification* - RTTI). Identificarea tipului în timpul rulării vă permite să scrieți cod portabil care poate determina tipul efectiv al obiectelor date în timpul rulării, chiar atunci când codul poate accesa numai un pointer sau o referință la acel obiect. Identificarea tipului în timpul rulării face posibil acest lucru, de exemplu, pentru a converti un pointer la o clasă de bază virtuală într-un pointer la tipul derivat al obiectului real. După cum ați învățat în secțiunea 1149, puteți utiliza operatorul *dynamic_cast* împreună cu identificare tipului în timpul rulării pentru a a face modelarea în timpul rulării.

UTILIZAREA OPERATORULUI TYPEID PENTRU IDENTIFICAREA TIPULUI ÎN TIMPUL RULĂRII

C/C++1156

După cum ați învățat în secțiunea 1155, identificarea tipului în timpul rulării permite programelor dumneavoastră să manipuleze pointerii și referințele într-o modalitate cu totul nouă. Mecanismul identificării tipului în timpul rulării vă permite, de asemenea, să verificați dacă un obiect este de un anumit tip și dacă două obiecte sunt de același tip. Puteți verifica obiectele cu operatorul *typeid*. Operatorul *typeid* determină tipul efectiv al argumentelor sale și returnează o referință la un obiect de tip *const typeinfo*, care descrie acel tip.

Puteți, de asemenea, să utilizați un nume de tip ca argument pentru *typeid*, iar *typeid* va returna o referință la un obiect *const typeinfo* pentru acel tip. Clasa *typeinfo* dispune de un operator == și de un operator != pe care îi puteți utiliza pentru a determina dacă două obiecte sunt de același tip. Clasa *typeinfo* dispune, de asemenea, de o funcție membru, *name*, care returnează un pointer la un șir de caractere care păstrează numele tipului. Trebuie să includeți fișierul antet *typeinfo.h* în cadrul programelor dumneavoastră pentru a accesa funcția *typeid*. Forma generală a funcției *typeid* este prezentată în continuare:

```
#include<typeinfo.h>
const typeinfo typeid(obiect);
```

În cadrul formei generalizate, *obiect* corespunde obiectului al cărui tip doriți ca *typeid* să îl returneze. Atunci când aplicați *typeid* unui pointer de clasă de bază a unei clase polimorfice, *typeid* va returna automat tipul obiectului la care *indică pointerul*, inclusiv orice clasă derivată din clasa de bază. Pentru a înțelege mai bine prelucrările pe care le efectuează *typeid*, studiați următorul program, *typeid_1.cpp*:

```
#include <iostream.h>
#include <typeinfo.h>

class A { };
class B : A { };

void main(void)
  {
```

```
  char C;
  float X;
  // UTILIZEAZA typeinfo::operator ==() PENTRU A COMPARA
  if (typeid( C ) == typeid( X ))
    cout << "C si X sunt de acelasi tip." << endl;
  else
    cout << "C si X NU sunt de acelasi tip." << endl;
  // UTILIZEAZA true SI false PENTRU COMPARATIE LEXICALA
  cout << typeid(int).name();
  cout << " inaintea lui " << typeid(double).name() << ": " <<
    (typeid(int).before(typeid(double)) ? true : false) << endl;
  cout << typeid(double).name();
  cout << " inaintea lui " << typeid(int).name() << ": " <<
    (typeid(double).before(typeid(int)) ? true : false) << endl;
  cout << typeid(A).name();
  cout << " inaintea lui " << typeid(B).name() << ": " <<
    (typeid(A).before(typeid(B)) ? true : false) << endl;
}
```

Programul *typeid_1.cpp* declară două clase, o clasă de bază (A) şi o clasă derivată (B). Atunci când programul începe execuţia, el defineşte două variabile, una de tip *char* şi una de tip *float*. Programul testează apoi tipurile atât al lui *X*, cât şi al lui *C*. Dacă ele sunt de acelaşi tip, programul va afişa un mesaj care afirmă aceasta pe ecran; dacă nu, programul va afişa un mesaj care afirmă ca nu sunt de acelaşi tip. Desigur, deoarece *char* şi *float* nu sunt de acelaşi tip, programul va afişa mesajul „C şi X NU sunt de acelasi tip.". Programul utilizează apoi membrul *before* pentru a compara anumite tipuri de bază cu o comparaţie lexicală. O comparaţie lexicală este o comparaţie bazată pe alfabet: a este înaintea lui b, c este înaintea lui d şi după b şi aşa mai departe. De aceea, atunci când programul compară *int* cu *double*, el va returna *false* când verifică dacă *double* este după *int*, dar *true* când verifică dacă *double* este înaintea lui *int*. În sfârşit, programul testează dacă A este înaintea lui B. Când compilaţi şi executaţi programul *typeid_1.cpp*, el va afişa următoarele pe ecran:

```
C si X NU sunt de acelasi tip.
int inaintea lui double: 0
double inaintea lui int: 1
A inaintea lui B: 1
C:\>
```

1157 CLASA TYPEINFO



După cum aţi învăţat în secţiunea 1156, funcţia *typeid* returnează o valoare de tip *const typeinfo*. Valoarea *const typeinfo* este un şir de caractere care reprezintă informaţii despre tipul clasei. Tabelul 1157 listează valorile returnate posibile pentru funcţia *typeid*.

Valoare	Semnificații
[ARRAY]	Valoarea este o matrice. Funcția *typeid* returnează întotdeauna [ARRAY] împreună cu un alt cuvânt cheie.
Numele clasei	Numele clasei definite de utilizator pentru obiect.
[INTEGER]	Obiectul este un întreg sau un întreg *long*.
[NULL]	Obiectul este o valoare *NULL* (în general, un pointer *NULL*).
[REAL]	Obiectul reprezintă un obiect *float, double* sau *long double*.
[STRING]	Obiectul este un șir de caractere.
[UNINITIALIZED]	Obiectul este neinițializat (în general, un pointer la o clasă polimorfică).

Tabelul 1157 *Posibilele valori returnate ale funcției* ***typeid.***

Clasa *typeinfo* definește, de asemenea, patru membri publici, în plus față de funcția *typeid*. Veți implementa aceste funcții membre în cadrul programelor dumneavoastră, ca mai jos:

```
bool operator==(const type_info &obiect) const;
bool operator!=(const type_info &obiect) const;
bool before(const type_info &obiect) const;
const char *name(void) const;
```

După cum ați învățat în secțiunea 1156, operatorii supraîncărcați == și != vă permit să comparați tipurile pe care *typeid* le returnează. Compilatorul va utiliza mai întâi funcția *before* în interior. Aceasta va returna *true* dacă obiectul apelant este înaintea obiectului *obiect* în ordine alfabetică (o listă derivată de compilator). Nu va returna informații privind ierarhia claselor sau alte tipuri utile de informații. Funcția *name* returnează un pointer la numele tipului. Următorul program, *typeid_2.cpp*, utilizează funcția *name* pentru a returna informații suplimentare dintr-o acțiune *typeid*:

```
#include <iostream.h>
#include <typeinfo.h>

class Baza {
   int a, b;
   virtual void func(void) {};
};
class Derivata1: public Baza {
   int i, j;
};
class Derivata2: public Baza {
   int k;
};
void main(void)
  {
   int i;
   Baza *p, obiectbaza;
   Derivata1 obiect1;
```

```
    Derivata2 obiect2;
    cout << "Typeid al lui i este: ";
    cout << typeid(i).name() << endl;
    p = &obiectbaza;
    cout << "p indica in mod curent catre un obiect de tip: ";
    cout << typeid(*p).name() << endl;
    p = & obiect1;
    cout << "p indica acum catre un obiect de tip: ";
    cout << typeid(*p).name() << endl;
    p = & obiect2;
    cout << "p indica in final catre un obiect de tip: ";
    cout << typeid(*p).name() << endl;
}
```

Programul *typeid_2.cpp* definește o singură clasă de bază și două clase derivate. Atunci când programul începe execuția, el declară două variabile de tipurile derivatelor, o variabilă de tipul bazei, un pointer la tipul bazei și o variabilă întreagă simplă.

Apoi, programul verifică *typeid* al variabilei întregi și returnează numele său în text. După returnarea informației despre tipul simplu *int* al variabilei, programul manipulează pointerul *p* pentru a indica fiecare dintre cele trei clase personalizate. De fiecare dată când tipul indicat de *p* se modifică, programul afișează informații despre noul nume al tipului. Atunci când compilați și executați programul *typeid_2.cpp*, el va afișa următoarea ieșire:

```
Typeid al lui i este:
p indica in mod curent catre un obiect de tip: Baza
p indica acum catre un obiect de tip: Derivata1
p indica in final catre un obiect de tip: Derivata2
C:\>
```

1158 CUVÂNTUL CHEIE MUTABLE



După cum ați învățat, C++ adaugă noi specificatori la declarațiile variabilelor, cum ar fi specificatorul *long double*, pe care puteți să-l utilizați în cadrul programelor dumneavoastră. În plus față de noile tipuri simple de date, programele dumneavoastră pot utiliza cuvântul cheie *mutable* cu o variabilă de orice tip pentru a face variabila modificabilă, deși ea se află într-o expresie care conține atributul *const*. De exemplu, următoarea declarație face ca *j* să fie o variabilă *mutable int*:

```
class UnExemplu {
    mutable int j;
}
```

Programele pot declara numai membri de date ai claselor ca *mutable*. Nu se poate utiliza cuvântul cheie *mutable* cu numele *static* sau *const*. Scopul cuvântului cheie *mutable* este specificarea acelor date membre pe care funcțiile membre *const* le pot modifica, deoarece o funcție membru *const*, în mod normal, nu poate modifica datele membre.

UTILIZAREA CUVÂNTULUI CHEIE MUTABLE ÎNTR-O CLASĂ

C/C++1159

După cum ați învățat în secțiunea 1158, programele dumneavoastră pot utiliza cuvântul cheie *mutable* în cadrul definiției unei clase pentru a face modificabilă o variabilă membru, deși ea se află într-o expresie cu atributul *const*, chiar atunci când obiectul căruia îi este membru este *const*. Puteți utiliza cuvântul cheie *mutable* numai cu membri într-o clasă *const*. Pentru a înțelege mai bine utilizarea cuvântului cheie *mutable*, studiați următorul program, *mutable.cpp*, care declară doi membri *mutable* în cadrul unei clase:

```
#include <iostream.h>

class Alpha {
  mutable int nr;
  mutable const int* iptr;
public:
  Alpha(void) {nr = 0;}
  int func1(int i = 0) const { // Promite ca nu modifica
                               // argumentele const.
    nr = i++; // Dar nr poate fi modificat.
    iptr = &i;
    cout << "i este: " << *iptr << endl;
    return nr;
    }
  void arata_nr(void) { cout << "Nr este: " << nr << endl;}
};

void main(void)
  {
    Alpha a;
    a.arata_nr();
    a.func1(10);
    a.arata_nr();
  }
```

În loc să lase membrul *nr* nemodificat, așa cum promite modificatorul *const* din declarația funcției *func1*, cuvântul cheie *mutable* permite funcției membru să modifice membrul *nr*. În loc să mențină valoarea inițială 0, *nr* are valoarea 10 la încheierea programului, deoarece *nr* este o valoare *mutable*. Atunci când compilați și executați programul *mutable.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Nr este: 0
i este: 11
Nr este: 10
C:\>
```

1160 OBSERVAȚII ÎN LEGĂTURĂ CU CUVÂNTUL CHEIE MUTABLE



După cum ați învățat, programele dumneavoastră pot utiliza cuvântul cheie *mutable* pentru a suprapune atributul *const* aplicat unui membru al unei clase. În timp ce cuvântul cheie *mutable* vă furnizează mijloace pentru un control mai bun asupra părților unei clase care trebuie să rămână nemodificată și asupra celor care sunt modificabile, cuvântul cheie *mutable* poate, de asemenea, să introducă erori semnificative și greu de urmărit. De exemplu, în programul *mutable.cpp* prezentat în secțiunea 1159, clasa *Alpha* a declarat variabila *nr* ca *mutable*. Apoi, clasa a definit o funcție *const*, care ar fi trebuit să nu modifice valorile nici unei variabile pe care o utilizează. În schimb, deoarece clasa a declarat variabila *nr* cu cuvântul cheie *mutable*, funcția *func1* modifică valoarea membrului *a.nr* la 10.

Trebuie să aveți multă grijă la utilizarea cuvântului cheie *mutable* în programele dumneavoastră, astfel încât să nu creați erori greu de urmărit. Ca multe dintre caracteristicile introduse de C++, decizia dumneavoastră de a folosi sau nu cuvântul cheie *mutable* depinde de necesitățile de claritate. Programul *mutable.cpp* din secțiunea precedentă ilustrează importanța clarității prin aceea că el este mult mai puțin clar utilizând cuvântul cheie *mutable*, decât ar fi neutilizându-l.

1161 PREZENTAREA TIPULUI BOOL DE DATE



Pe parcursul secțiunilor precedente ați utilizat date de tip *int* în cadrul comparațiilor logice. Cele mai noi compilatoare de C++ acceptă tipul *bool* de date pentru prelucrarea datelor booleene. Tipul de date *bool* acceptă valorile adevărat (*true*) și fals (*false*). Veți declara și utiliza variabile de tip *bool* cum arătăm mai jos:

```
bool variabila_logica;
variabila_logica = true;
```

Deoarece tipul de date *bool* acceptă cuvintele cheie *true* și *false* ca *rvalue*, nu trebuie să definiți variabilele numite *true* sau *false* în cadrul programelor dumneavoastră.

1162 UTILIZAREA TIPULUI DE DATE BOOL



După cum ați învățat în secțiunea 1161, C++ acceptă noul tip de date *bool*, pe care programele dumneavoastră îl pot folosi pentru a păstra informații logice. Noul tip de date *bool* utilizează două noi cuvinte cheie ale limbajului C++, *true* și *false*. Cuvintele cheie *true* și *false* corespund valorilor adevărat (1) și fals (0) pentru întregi. Puteți să utilizați tipul de date *bool* în cadrul programelor dumneavoastră pentru a face codul mai clar. De exemplu, puteți atribui rezultatul unui test logic unui tip de date *bool*, cum arătăm în continuare:

```
bool rezultat;
rezultat = (A && B);
```

Următorul program, *bool_fun.cpp*, utilizează o funcție care returnează o valoare de tip *bool*:

```
#include <iostream.h>
bool func(void)
```

```
  {     // Functia returneaza un tip bool
    return false;
    // return NULL; // NULL este convertit la false Boolean
  }
void main(void)
  {
    bool val = false; // variabila booleana
    int i = 1; // i nu este nici true-Boolean, nici false-Boolean
    int g = 3;
    int *iptr = 0;   // pointer nul
    float j = 1.01;  // j nu este nici true-Boolean, nici
                     // false-Boolean
    if (i == true)
      cout << "Adevarat: valoarea este 1" << endl;
    if (i == false)
      cout << "Fals: valoarea este 0" << endl;
    if (g)
      cout << "g este adevarat.";
    else
      cout << "g este fals.";
    // Testeaza pointerul
    if (*iptr == false)
      cout << "Pointer nevalid." << endl;
    if (*iptr == true)
      cout << "Pointer valid." << endl;
    // Pentru a testa valoarea de adevar a lui j, il modelam
    // la tipul bool.
    if (bool(j) == true)
      cout << "j Boolean este adevarat." << endl;
    // Valoarea returnata de functia de testare Booleana
    val = func();
    if (val == false)
      cout << "func() a returnat false.";
    if (val == true)
      cout << "func() a returnat true.";
  }
```

În plus față de utilizarea unei funcții *bool*, programul efectuează o serie de comparații utilizând valori *bool* în loc de întregi și convertește o valoare *int* la un rezultat *bool*. Atunci când compilați și executați programul *bool_fun.cpp*, ecranul dumneavoastră va afișa următorul rezultat (al doilea rând conține un mesaj de eroare prin care se anunță că g nu are o valoare de adevăr cunoscută):

```
Adevarat: valoarea este 1
Unknown truth value for g.
Pointer nevalid.
j Boolean este adevarat.
```

```
func() a returnat false.
C:\>
```

1163 CREAREA UNUI TIP DE SIR

C/C++

După cum aţi învăţat, C++ implementează şiruri de caractere ca matrice terminate în caracterul *NULL* şi nu ca un tip aparte de date. Majoritatea versiunilor curente de C++ implementează biblioteca standard *<string.h>* care creează un tip aparte de date pentru şiruri. Pentru a înţelege mai bine modul în care se implementează şi se manipulează toate clasele în programele dumneavoastră, veţi învăţa să creaţi un tip personalizat de şir de caractere. În următoarele câteva secţiuni, veţi utiliza ceea ce aţi învăţat din secţiunile anterioare pentru a crea un tip de date şir de caractere dezvoltat.

Primul pas în crearea oricărui nou tip este determinarea necesităţilor pe care doriţi să le satisfacă acesta. Pe scurt, trebuie să *definiţi* tipul. În secţiunea 1164 veţi defini caracteristicile tipului dumneavoastră propriu şir de caractere – operatorii pe care programele dumneavoastră îi pot utiliza cu tipul, funcţiile membre pe care tipul le acceptă şi aşa mai departe.

1164 DEFINIREA CARACTERISTICILOR TIPULUI ŞIR DE CARACTERE

C/C++

După cum ştiţi,atunci când lucraţi cu o matrice de tip *char* în C++, lucraţi de fapt cu câteva elemente char pe care C++ le stochează în succesiune în memoria calculatorului. De exemplu, definiţia prezentată mai jos creează în realitate 32 de elemente separate de tip *char* în cadrul matricei *exemplu*:

```
char exemplu[ ] = "Jamsa's C/C++ Programmer's Bible";
```

În general, ar fi semnificativ mai uşor să atribuiţi valoarea direct şirului *exemplu*, ca mai jos:

```
Siruri exemplu = "Jamsa's C/C++ Programmer's Bible";
```

Când creaţi tipul în modalitatea prezentată de al doilea exemplu, puteţi să atribuiţi mai târziu şirul la un alt şir, ca mai jos:

```
Siruri exemplu1, exemplu2;
exemplu1 = "Jamsa's C/C++ Programmer's Bible";
exemplu2 = exemplu1;
```

Desigur, fragmentul de cod precedent, care utilizează clasa *Siruri*, este mai uşor de înţeles decât următorul fragment de cod:

```
Char exemplu[ ] = "Jamsa's C/C++ Programmer's Bible";
Char exemplu2[256];
strcpy(exemplu1, exemplu);
```

În plus, clasa dumneavoastră *Siruri* ar trebui să vă permită să utilizaţi operatorul +, faţă de funcţia *strcat* pe care aţi învăţat-o în secţiunile anterioare, pentru a concatena două şiruri de caractere, cum arătăm mai jos:

```
Siruri exemplu, exemplu2;
exemplu = "Jamsa's C/C++ ";
exemplu2 = "Programmer's Bible";
exemplu = exemplu + exemplu2;
```

Clasa dumneavoastră *Siruri* ar trebui, de asemenea, să vă permită adăugarea la şir a unui şir de caractere de tip diferit faţă de *Siruri*, cum arătăm în continuare:

```
exemplu = exemplu + "Acesta este un exemplu.";
```

În mod asemănător, clasa dumneavoastră *Siruri* ar trebui să vă permită scoaterea unui subşir din cadrul şirului, utilizând operatorul de scădere. În sfârşit, clasa dumneavoastră *Siruri* ar trebui să vă permită compararea şirurilor utilizând operatori relaţionali, cum ar fi operatorii „mai mare decât" şi „mai mic decât", în loc de funcţia *strcmp*. Cu alte cuvinte, următorul fragment de cod ar trebui să compare în mod corespunzător *exemplu1* cu *exemplu2*:

```
Siruri exemplu1, exemplu2;
exemplu1 = "Jamsa's C/C++ Programmer's Bible";
exemplu2 = "Jamsa's C/C++ Programmer's Book";
if(exemplu2 > exemplu1)
    cout << exemplu2 << " este mai mare decat " << exemplu1;
```

Acum, că aţi înţeles unele operaţii de bază pe care veţi dori să le îndeplinească clasa dumneavoastră de tip şir, ar trebui să începeţi să implementaţi clasa *Siruri*. În secţiunea 1165 veţi crea declaraţia pentru clasa dumneavoastră *Siruri*.

CREAREA CLASEI SIRURI

C/C++1165

După cum aţi învăţat, puteţi utiliza o clasă C++ pentru a crea propriul dumneavoastră tip şir, în loc de a lucra în continuare cu matrice de tip *char*. În secţiunea 1164 aţi revăzut câteva dintre operaţiile de bază pe care aveţi posibilitatea să le efectuaţi cu clasa dumneavoastră pentru şiruri. De fapt, clasa *Siruri* pe care o veţi defini în această secţiune este construită integral din definiţia pe care aţi creat-o pentru clasa *Siruri* din secţiunea 1164. Trebuie să remacaţi că definiţia clasei *Siruri* utilizează funcţii *friend* pentru a supraîncărca numai câţiva operatori şi în primul rând, utilizează funcţii membre pentru supraîncărcarea operatorului relaţional. Însă, aţi putea de asemenea să scrieţi clasa utilizând funcţiile *friend* şi să executaţi de asemenea aceleaşi prelucrări cu definiţia curentă. CD-ROM-ul însoţitor al acestei cărţi include fişierul *strings.cpp*, care conţine definiţia clasei pentru clasa *strings*:

```
class Siruri {
    char *p;
    int dimensiune;
public:
    Siruri(char *sir);
    Siruri(void);
    Siruri(const Siruri &obiect); // Copie a constructorului
    ~Siruri(void) {delete [ ] p;}

    friend ostream &operator<<(ostream &flux, Siruri &obiect);
    friend istream &operator>>(istream &flux, Siruri &obiect);
```

```
    Siruri operator=(Siruri &obiect); // atribuie un obiect
                                      // Siruri
    Siruri operator=(char *s); // atribuie un sir intre
                               // ghilimele
    Siruri operator+(Siruri &obiect); // concateneaza un obiect
                                      // Siruri
    Siruri operator+(char *s); // concateneaza un sir intre
                               // ghilimele
    friend Siruri operator+(char *s, Siruri &obiect);
       /* concateneaza un sir intre ghilimele cu un obiect
       /* Siruri */
    Siruri operator-(Siruri &obiect); // scade un obiect Siruri
    Siruri operator-(char *s); // scade un sir intre ghilimele
/* operatori relationali intre obiecte Siruri. Notati ca
   operatorii  ar putea la fel de usor sa returneze bool, in
   loc de int */

    int operator==(Siruri &obiect) {return !strcmp(p, obiect.p);}
    int operator!=(Siruri &obiect) {return strcmp(p, obiect.p);}
    int operator<(Siruri &obiect) {return strcmp(p, obiect.p) < 0;}
    int operator>(Siruri &obiect) {return strcmp(p, obiect.p) > 0;}
    int operator<=(Siruri &obiect) {return strcmp(p, obiect.p)
      <= 0;}
    int operator>=(Siruri &obiect) {return strcmp(p, obiect.p)
      >= 0;}
/* operatori relationali intre obiecte Siruri si un sir de
   caractere intre ghilimele. Notati ca operatorii ar putea la
   fel de usor sa  returneze bool, in loc de int */

    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}

    int dimsir(void) {return strlen(p);}
               // returneaza dimensiunea sirului
    void creatsir(char *s) (strcpy(s, p);}
               // face sir intre ghilimele din obiectul Siruri
    operator char *(void) {return p;} // conversie in char
}
```

Majoritatea funcțiilor membre din cadrul clasei *Siruri* sunt relativ explicabile prin ele însele, deoarece sunt construiți în baza tehnicilor de supraîncărcare a operatorilor pe care le-ați învățat în secțiunile anterioare. Însă, trebuie să observați în special că această clasă conține numai doi membri privați, *p* și *dimensiune*. Așa cum veți învăța în secțiunea 1166, atunci când creați un obiect *Siruri*, funcția constructor va utiliza *new* pentru a aloca memorie

dinamică și a plasa un pointer către memorie în membrul *p*. Membrul *dimensiune* va păstra un întreg care reprezintă lungimea șirului.

SCRIEREA CONSTRUCTORILOR PENTRU CLASA SIRURI

C/C++1166

După cum ați învățat în declarația clasei *Siruri* pe care ați creat-o în secțiunea 1165, clasa *Siruri* acceptă trei constructori diferiți: un constructor neinițializat, un constructor care așteaptă un șir de caractere între ghilimele pentru inițializare și un constructor care așteaptă un alt obiect *Siruri* pentru inițializare. După cum veți vedea, constructorul efectuează în esență aceeași prelucrare în cele două cazuri din urmă. Însă, trebuie să creați un constructor care să permită programului dumneavoastră să conțină toate posibilitățile (exact cum ați văzut în cazul operatorilor supraîncărcați din secțiunea 1165). CD-ROM-ul care însoțește această carte conține fișierul *strings.cpp*, care include declarația din secțiune 1165 și codul din această secțiune, ca mai jos:

```
Siruri::Siruri(void)
  {
    dimens = 1;
    p = new char[dimens];
    if(!p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
      *p = '\0';
  }
Siruri::Siruri(char *sir)
  {
    dimens = strlen(sir) + 1;
    p = new char[dimens];
    if(!p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
      strcpy(p, sir);
  }
Siruri::Siruri(const Siruri &obiect)
  {
    dimens = obiect.dimens;
    p = new char[dimens];
    if(!p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
```

```
    strcpy(p, obiect.p);
}
```

După cum puteți vedea, fiecare constructor alocă memorie, creează o matrice de tip *char* de dimensiunea cerută și inițializează *p* astfel încât să indice la începutul matricei. Atunci când programele dumneavoastră distrug obiectul *Siruri*, funcția destructor inline eliberează pur și simplu memoria indicată de *p*.

1167 INTRĂRI/IEȘIRI CU CLASA SIRURI

C/C++

În secțiunea 1165 ați creat definiția de bază pentru clasa *Siruri*. Definiția cuprinde referințele la două funcții *friend* care supraîncarcă operatorii de inserare și de extragere, cum arătăm mai jos:

```
friend ostream &operator<<(ostream &flux, Siruri &obiect);
friend istream &operator>>(istream &flux, Siruri &obiect);
```

Deorece intrarea și ieșirea sunt cele mai comune operații pe care programele dumneavoastră le vor efectua cu șiruri de caractere, trebuie să implementați o versiune supraîncărcată în cadrul clasei. Fluxul de ieșire manipulează cu ușurință obiectele *Siruri*, transmițând informația direct fluxului. Observați că extractorul primește obiectul *Siruri* prin referință. Deoarece obiectele *Siruri* pot fi, din punct de vedere teoretic destul de mari, ele vor îmbunătăți performanțele programelor dumneavoastră de a transmite obiectului *Siruri* ca pointer, în loc de valoare, după cum urmează:

```
ostream &operator<<(ostream &flux, Siruri &obiect)
  {
    flux << obiect.p;
    return flux;
  }
```

1168 SCRIEREA FUNCȚIILOR DE ATRIBUIRE PENTRU CLASA SIRURI

C/C++

Așa cum ați văzut în secțiunea 1165, scrierea funcțiilor de atribuire pentru clasa *Siruri* este repetitivă, similară scrierii funcțiilor constructor ale clasei. Cei doi operatori supraîncărcați trebuie să atribuie obiectului *Siruri* fie valoarea altui obiect *Siruri*, fie valoarea unui șir de caractere între ghilimele și trebuie să opereze în cele două situații în moduri ușor diferite. După cum puteți vedea, în ambele cazuri operatorul de atribuire supraîncărcat șterge memoria curent alocată pentru *p*, creează o nouă memorie pentru *p* și apoi atribuie noua valoare la noua memorie. Singura diferență semnificativă între cele două funcții este aceea că prima primește un operand de tip *Siruri*, iar a doua primește un operand de tip *char **, ca mai jos:

```
Siruri Siruri::operator=(Siruri &obiect)
  {
    Siruri temp(obiect.p);
    if(obiect.dimens > dimens)
      {
```

```
      delete p;
      p = new char[obiect.dimens];
      dimens = obiect.dimens;
      if(!p)
        {
          cout << "Eroare la alocare!" << endl;
          exit(1);
        }
    }
    strcpy(p, obiect.p);
    strcpy(temp.p, obiect.p);
    return temp;
}
Siruri Siruri::operator=(char *s)
  {
    int lungime = srtlen(s) + 1;
    if(dimens < lungime)
    {
      delete p;
      p = new char[lungime];
      dimens = lungime;
      if(!p)
        {
          cout << "Eroare la alocare!" << endl;
          exit(1);
        }
    }
    strcpy(p, s);
    return *this;
}
```

Ambele versiuni supraîncărcate ale operatorului = utilizează funcția *strcpy* pentru a plasa valoarea șirului de tip *rvalue* în șirul de tip *lvalue*. Deoarece atribuirea modifică în mod direct șirul de tip *lvalue*, ambele funcții utilizează pointerul *this* pentru a atribui noua valoare obiectului *Siruri* din partea stângă.

SUPRAÎNCĂRCAREA OPERATORULUI + PENTRU A CONCATENA OBIECTE SIRURI

C/C++1169

Așa cum ați determinat în secțiunea 1164, clasa dumneavoastră *Siruri* trebuie să vă permită utilizarea operatorului + pentru a concatena două obiecte *Siruri*. La fel cum ați supraîncărcat operatorul de atribuire pentru a permite clasei Siruri operarea atât cu alt obiect *Siruri*, cât și cu un șir de caractere între ghilimele, tot așa va trebui să supraîncărcați operatorul + pentru a controla ambele situații. Însă, pentru că operatorul + este un operator binar și nu unul unar, trebuie să creați trei versiuni supraîncărcate ale sale pentru a controla toate situațiile de concatenare: una care operează cu două obiecte *Siruri*, alta care operează cu un obiect *Siruri* și un șir de caractere între ghilimele la dreapta operatorului și o a treia care operează

cu un şir de caractere între ghilimele şi un obiect *Siruri* aflat la dreapta operatorului. După cum aţi învăţat, puteţi utiliza un şablon de funcţie *friend* pentru a evita rescrierea aceluiaşi algoritm de trei ori, dar nu veţi proceda astfel în acest caz, pentru că este util să înţelegeţi fiecare situaţie pe care codul dumneavoastră trebuie să o controleze. La fel ca în secţiunea anterioară, CD-ROM-ul care însoţeşte această carte conţine în cadrul fişierului *strings.cpp* codul pentru implementarea funcţiilor de supraîncărcare a operatorului +. Veţi implementa aceste funcţii aşa cum arătăm în continuare:

```
Siruri Siruri::operator+(Siruri &obiect)
  {
    int lungime;
    Siruri temp;

    delete temp.p;
    lungime = strlen(obiect.p) + strlen(p) + 1;
    temp.p = new char[lungime];
    temp.dimens = lungime;
    if(!temp.p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
    strcpy(temp.p, this.p);
    strcat(temp.p, obiect.p);
    return temp;
  }
Siruri Siruri::operator+(char *s)
  {
    int lungime;
    Siruri temp;

    delete temp.p;
    lungime = strlen(s) + strlen(p) + 1;
    temp.p = new char[lungime];
    temp.dimens = lungime;
    if(!temp.p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
    strcpy(temp.p, this.p);
    strcat(temp.p, s);
    return temp;
  }
Siruri operator+(char *s, Siruri &obiect)
  {
    int lungime;
    Siruri temp;
```

```
    delete temp.p;
    lungime = strlen(s) + strlen(p) + 1;
    temp.p = new char[lungime];
    temp.dimens = lungime;
    if(!temp.p)
      {
        cout << "Eroare la alocare!" << endl;
        exit(1);
      }
    strcpy(temp.p, s);
    strcat(temp.p, obiect.p);
    return temp;
  }
```

Observați că a treia funcție de supraîncărcare este o funcție *friend* și nu un operator membru supraîncărcat. Așa cum ați învățat, atunci când utilizați un operator membru, el transmite în mod explicit obiectul de la stânga – ceea ce va provoca o eroare dacă obiectul de la stânga nu este de tipul corespunzător clasei. Atunci când adăugați un obiect *Siruri* la un șir de caractere între ghilimele, trebuie să transmiteți explicit șirul și să adăugați obiectul la valoarea șirului citat între ghilimele.

ELIMINAREA UNUI ȘIR DE CARACTERE DIN CADRUL UNUI OBIECT SIRURI

C/C++1170

O funcție utilă pentru șiruri pe care veți dori să o adăugați obiectului dumneavoastră *Siruri* este extragerea unui subșir. Atunci când obiectul *Siruri* implementează operația de extragere a unui subșir, el elimină toate aparițiile respectivului subșir din cadrul obiectului *Siruri*. Pentru a înțelege mai bine modul în care *Siruri* implementează operația de extragere a subșirului, analizați următorul fragment de cod:

```
exemplu = "Jamsa's C/C++ Programmer's Bible";
exemplu = exemplu - "C";
// exemplu este acum egal cu "Jamsa's /++ Programmer's Bible"
```

Spre deosebire de cele trei funcții supraîncărcate pe care le-ați implementat pentru supraîncărcarea operatorului +, veți utiliza numai două funcții supraîncărcate pentru supraîncărcarea operatorului – (pentru că nu veți extrage obiectul *Siruri* dintr-un șir de caractere). CD-ROM-ul care însoțește această carte conține codul de implementare în cadrul fișierului *strings.cpp*, ca mai jos:

```
Siruri Siruri::operator-(Siruri &subsir)
  {
    Siruri temp(p);
    char *s1;
    int i,j;
    s1 = p;
    for(i=0; *s1; i++)
      {
```

```
      if(*s1!=*subsir.p)
        {
          temp.p[i] = *s1;
          s1++;
        }
      else
        {
          for(j=0; subsir.p[j]==s1[j] && subsir.p[j]; j++)
            ;
          if(!subsir.p[j])
            {
              s1 += j;
              i--;
            }
          else
            {
              temp.p[i] = *s1;
              s1++;
            }
        }
    }
  temp.p[i] = '\0';
  return temp;
 }
Siruri Siruri::operator-(char *subsir)
 {
  Siruri temp(p);
  char *s1;
  int i,j;
  s1 = p;
  for(i=0; *s1; i++)
    {
      if(*s1!=*subsir)
        {
          temp.p[i] = *s1;
          s1++;
        }
      else
        {
          for(j=0; subsir[j]==s1[j] && subsir[j]; j++)
            ;
          if(!subsir[j])
            {
              s1 += j;
              i--;
            }
```

```
            else
            {
               temp.p[i] = *s1;
               s1++;
            }
         }
      }
      temp.p[i] = '\0';
      return temp;
   }
```

Ambele funcții supraîncărcate copiază conținutul operandului din stânga într-o variabilă *temp*. Cum fiecare funcție copiază operandul din stânga, ele vor elimina orice apariție a subșirului pe care îl specifică operandul din partea dreaptă, în timpul prelucrării. Funcțiile de supraîncărcare a operatorului returnează apoi obiectul *Siruri* rezultat. Datorită modului în care ați definit până acum clasa *Siruri*, toate instrucțiunile următoare sunt valabile pentru utilizare cu operatorul de scădere a obiectului *Siruri*:

```
Siruri x("ABCABCD"), y("A");
Siruri z;
z = x - y; // z = "BCBCD"
```

SUPRAÎNCĂRCAREA OPERATORILOR RELAȚIONALI

C/C++1171

În secțiunile anterioare ați supraîncărcat mai mulți operatori pe care programele dumneavoastră pot să îi utilizeze eficient pentru a manipula șiruri de caractere. Un alt aspect important al puterii clasei *Siruri* este abilitatea sa de a compara cu ușurință două șiruri și de a returna un rezultat mai semnificativ decât -1, 0 sau 1 (valorile returnate ale funcției *strcmp*). Operatorii relaționali pentru clasa *Siruri*, care returnează astfel de valori, sunt simpli. Ei utilizează funcția *strcmp* asupra membrului *p* din partea stângă a operatorului și fie un alt obiect, fie un pointer *char* pentru a evalua partea dreaptă a operatorului. Datorită simplității acțiunii efectuate de operatorii relaționali, clasa *Siruri* definește toți operatorii relaționali inline în cadrul definiției clasei. Definițiile operatorilor relaționali se află în cadrul fișierului *strings.cpp* de pe CD-ROM și le prezentăm în continuare:

```
/* operatori relationali intre obiecte Siruri. Notati ca
   operatorii ar putea la fel de usor sa returneze bool in loc
   de int */
    int operator==(Siruri &obiect) {return !strcmp(p, obiect.p);}
    int operator!=(Siruri &obiect) {return strcmp(p, obiect.p);}
    int operator<(Siruri &obiect) {return strcmp(p, obiect.p) < 0;}
    int operator>(Siruri &obiect) {return strcmp(p, obiect.p) > 0;}
    int operator<=(Siruri &obiect) {return strcmp(p, obiect.p)
       <= 0;}
    int operator>=(Siruri &obiect) {return strcmp(p, obiect.p)
       >= 0;}
```

```
/* operatori relationali intre obiecte Siruri si un sir de
   caractere intre ghilimele. Notati ca operatorii ar putea la
   fel de usor sa returneze bool in loc de int */
    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}
```

Toate definițiile pentru operatorii relaționali din cadrul clasei *Siruri* presupun că veți compara un obiect *Siruri* cu un alt obiect *Siruri* sau cu un șir de caractere între ghilimele. Dacă doriți ca programele dumneavoastră să poată compara șiruri de caractere între ghilimele cu un obiect *Siruri*, trebuie să definiți un alt set de funcții *friend* pentru a supraîncărca din nou operatorii relaționali. De exemplu, trebuie să declarați a treia supraîncărcare a operatorului ==, ca mai jos:

```
friend Siruri operator == (char * s, Siruri &obiect);
```

În afara definiției clasei, trebuie să creați o funcție de supraîncărcare a operatorilor, cum arătăm mai jos:

```
Siruri operator==( char * s, Siruri &obiect)
  {
  return !strcmp(s, obiect.p);
  }
```

1172 DETERMINAREA DIMENSIUNII UNUI OBIECT SIRURI

C/C++

Una dintre cele mai comune activități pe care le veți executa cu șirurile de caractere este determinarea dimensiunilor lor curente. Deseori, veți utiliza dimensiunea curentă a unui șir pentru a efectua prelucrări suplimentare cu șirul, pentru a formata ieșirea și așa mai departe. Pentru a vă ajuta să determinați lungimea unui obiect *Siruri*, clasa *Siruri* vă pune la dispoziție următoarea funcție membru:

```
int dimsir(void) {return strlen(p);}    // returneaza dimensiunea
                                        // sirului
```

Atunci când invocați funcția membru *dimsir* asupra unui obiect *Siruri*, funcția membru returnează o valoare întreagă care reprezintă lungimea obiectului *Siruri*. Veți invoca funcția membru asupra obiectului *Siruri*, ca mai jos:

```
Siruri exemplu;
int x;
x = exemplu.dimsir();
```

CONVERSIA UNUI OBIECT SIRURI ÎNTR-O MATRICE DE CARACTERE

C/C++1173

Definiția clasei *Siruri* furnizează două funcții membre pe care programele dumneavoastră le pot utiliza pentru manipularea matricei de tip *char* în cadrul obiectului *Siruri*, fără a opera asupra obiectului însuși. Prima funcție, *creatsir*, copiază matricea de caractere dintr-un obiect *Siruri* într-o matrice de tip *char* standard. Cel mai adesea, veți folosi funcția *creatsir* pentru a obține un șir terminat în caracterul *NULL* din obiectul *Siruri*. Programele dumneavoastră vor utiliza funcția membru *creatsir* asupra obiectului *Siruri*, cum prezentăm mai jos:

```
Siruri exemplu = "Jamsa's C/C++ Programmer's Bible"
char tablou[256];
exemplu.creatsir(tablou);
```

Utilizarea funcției membru *creatsir* vă permite să alegeți între matricele de tip *char* și obiectele *Siruri* și, de asemenea, să le convertiți cu ușurință între ele.

UTILIZAREA OBIECTULUI SIRURI CA MATRICE DE CARACTERE

C/C++1174

În secțiunea 1173 ați învățat că definiția clasei *Siruri* conține două funcții membre care vă ajută să utilizați obiectele *Siruri* ca matrice de tip *char* în cadrul programelor dumneavoastră. Prima funcție membru, *creatsir*, copiază componenta de tip șir a obiectului *Siruri* într-o matrice de tip *char*. Cea de a doua funcție membru, o supraîncărcare a operatorului (), vă permite să utilizați în mod direct obiectul *Siruri* în cadrul oricărei funcții care așteaptă o matrice normală, terminată în caracterul *NULL*. De exemplu, următorul cod se va compila și executa corect:

```
Siruri x("salut");
puts(x);
```

Deoarece doriți să evitați permanenta supraîncărcare a oricărei funcții care utilizează sau așteaptă să utilizeze o matrice sau un pointer de tip *char*, supraîncărcarea operatorului (), astfel ca el să returneze o matrice *char*, în loc de informații despre obiectul *Siruri*, menține un cod mai clar și face obiectul *Siruri* mai util.

DEMONSTRAREA OBIECTULUI SIRURI

C/C++1175

Utilizarea clasei *Siruri* pe care ați creat-o în secțiunile precedente este surprinzător de simplă. După ce ați terminat definiția, utilizarea multor capacități ale obiectelor *Siruri* este un proces simplu. Următorul fragment de cod utilizează definiția clasei *Siruri* și demonstrează unele din multele sale capacități:

```
void main(void)
  {
    Siruri s1("Un program exemplu care utilizeaza obiecte sir.\n");
    Siruri s2(s1);
    Siruri s3;
```

```
char s[80];

cout << s1 << s2;
s3 = s1;
cout << s3;
s3.creatsir(s);
cout << "Convertit in sir: " << s;

s2 = "Acesta este un sir nou.";
cout << s2 << endl;

Siruri s4("Acesta este un sir nou, de asemenea.");
s1 = s2 + s4;
cout << s1 << endl;

if(s2==s3)
  cout << "Sirurile sunt egale." << endl;
if(s2!=s3)
  cout << "Sirurile nu sunt egale." << endl;
if(s1<s4)
  cout << "s1 este mai mic decat s4." << endl;
if(s1>s4)
  cout << "s1 este mai mare decat s4." << endl;
if(s1<=s4)
  cout << "s1 este mai mic sau egal cu s4." << endl;
if(s1>=s4)
  cout << "s1 este mai mare sau egal cu s4." << endl;
if(s2 > "ABC")
  cout << "s2 este mai mare decat 'ABC'" << endl << endl;

s1 = "unu doi trei unu doi trei\n";
s2 = "doi";
cout << "Sir initial: " << s1;
cout << "Sir dupa extragerea lui doi: ";
s3 = s1 - s2;
cout << s3;

cout << endl;
s4 = "Jamsa's C/C++ ";
s3 = s4 + "Programmer's Bible\n";
cout << s3;
s3 = s3 - "C/C++";
s3 = "Aceasta este " + s3;
cout << s3;

cout << "Introduceti un sir: ";
cin >> s1;
cout << s1 << endl;
cout << "s1 este de " << s1.strsize() << " caractere." << endl
puts(s1);
```

```
    s1 = s2 = s3;
    cout << s1 << s2 << s3;
    s1 = s2 = s3 = "Program incheiat.\n";
    cout << s1 << s2 << s3;
}
```

Funcția *main* aflată în cadrul fragmentului de cod precedent apare în interiorul programului *use_str.cpp* de pe CD-ROM-ul care însoțește această carte.

CREAREA UNUI ANTET PENTRU CLASA SIRURI

C/C++1176

În secțiunile precedente ați creat o clasă *Siruri* cuprinzătoare și utilă. În cazul în care calculatorul dumneavoastră nu conține definiția clasei *string* sau dacă dumneavoastră considerați că vă place mai mult clasa *Siruri*, puteți pune definiția clasei într-un fișier antet pentru a nu mai ocupa atât spațiu în interiorul programelor dumneavoastră. Pentru a converti clasa *Siruri* într-un fișier antet, ștergeți funcția *main* și toate componentele sale din programul *util_sir.cpp (use_str.cpp)*. Apoi salvați codul rămas într-un fișier antet denumit *siruri.h (strings.h)*. Salvați fișierul antet în cadrul directorului *include* al compilatorului dumneavoastră. Mai târziu, când veți dori să utilizați clasa *Siruri* în cadrul programelor dumneavoastră, veți putea apela fișierul antet din cadrul codului programului, ca mai jos:

```
#include <siruri.h>
```

ALT EXEMPLU DE CLASĂ DE TIP SIRURI

C/C++1177

În secțiunea 1175, ați scris un program care a implementat propria dumneavoastră clasă *Siruri*. În secțiunea 1176, ați învățat cum să creați un fișier antet pentru clasa dumneavoastră *Siruri*. Pentru a înțelege mai bine puterea limbajului C++ de a crea fișiere antet personalizate, v-ar fi de folos să scrieți un alt program care utilizează noul fișier antet *siruri.h*. Programul din această secțiune, *test_fis.cpp* acceptă un singur parametru în linia de comandă, care corespunde numelui unui program executabil pe care doriți să-l găsiți. Programul va încerca apoi să deschidă un fișier program din unitatea locală, care are același nume. Dacă programul reușește, el vă va face cunoscut faptul că fișierul din unitatea locală există, iar dacă el va eșua, vă va informa că acel fișier nu există. Veți implementa programul *test_fis.cpp* ca mai jos:

```
#include "siruri.h"
#include <iostream.h>
#include <fstream.h>

//extensiile fisierelor executabile

char ext[3][4] = { "EXE", "COM", "BAT" };

void main(int argc, char *argv[ ])
  {
   Siruri fnume;
   int i;
```

```
    if(argc != 2)
      {
        cout << "Utilizare: fnume nume" << endl;
        exit(1);
      }
    fnume = argv[1];
    fnume = fnume + ".";
    for (i = 0; i<3; i++)
      {
        fnume = fnume + ext[i];
        cout << "Cautam " << fnume << " ";
        ifstream f.open(fnume);
        if(f)
          {
            cout << " - Exista" << endl;
            f.close();
          }
        else
            cout << " - Nu l-am gasit"<< endl;
        fnume = fnume - ext[i];
      }
  }
```

Este important în special să notați faptul că compilatorul acceptă apelul *f.open* chiar dacă el recepționează un obiect de tipul *Siruri* și nu un pointer la tipul *char*. Compilatorul acceptă apelul *f.open* deoarece prin supraîncărcarea operatorului *()* în cadrul clasei *Siruri* i-ați permis compilatorului să interpreteze acea informație ca fiind de tip *char** și nu un obiect de tipul *Siruri*. Atunci când compilați și executați programul *test_fis.cpp*, ecranul dumneavoastră va afișa următoarele:

```
C:\>test_fis test
test.EXE - Nu l-am gasit
test.OBJ - Nu l-am gasit
test.COM - Nu l-am gasit
C:\>test_fis test_fis
test_fis.EXE - Exista
test_fis.OBJ - Exista
test_fis.COM - Exista
```

1178 UTILIZAREA UNEI CLASE C++ PENTRU A CREA O LISTĂ DUBLU ÎNLĂNȚUITĂ

C/C++

În secțiunile de mai sus ați extins cunoștințele dumneavoastră despre modalitățile de creare și utilizare a propriilor dumneavoastră clase. În seria de exemple *Siruri* ați învățat cum se derivează propria clasă și cum se utilizează pe parcursul programului. O altă caracteristică utilă a claselor este capacitatea lor de a controla câteva reguli diferite pentru sprijinirea prelucrării de date. De exemplu, dacă vă amintiți secțiunea 749, acolo ați creat structura *IntrareLista* pentru a procesa o listă dublu înlănțuită, cum arătăm mai jos:

```
struct IntrareLista {
    int numar;
    struct IntrareLista *urmator;
    struct IntrareLista *precedent;
  } start, *nod;
```

Însă, pentru a efectua operații pe orice nod al listei (cum ar fi deplasarea prin listă), trebuie să accesați o serie de funcții exterioare. În următoarele secțiuni veți învăța modul în care se construiește o clasă *obiect_lista*, pe care o puteți folosi în cadrul programelor dumneavoastră pentru a menține informația despre liste dublu înlănțuite.

MEMBRII CLASEI CU LISTE DUBLU ÎNLĂNȚUITE

C/C++1179

În secțiunea 1178 ați rememorat pe scurt structura *IntrareLista* pe care ați creat-o anterior. Atunci când lucrați cu clasa *obiect_lista*, membrii vor fi puțin diferiți. Nu veți adăuga, efectiv, nici un fel de noi date membri; membrii pe care îi veți adăuga clasei dumneavoastră *obiect_lista* vor fi antetele pentru setul de funcții pe care programul dumneavoastră le va utiliza pentru a naviga mai bine și pentru a manipula lista înlănțuită. CD-ROM-ul care însoțește această carte cuprinde membrii clasei *list_object* în cadrul fișierului *dblinkcl.cpp*, ca mai jos:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class obiect_lista {
  public:
    char info;
    obiect_lista *urmator;
    obiect_lista *precedent;
    obiect_lista(void) {
      info = 0;
      urmator = NULL;
      precedent = NULL;
    }
    obiect_lista *redaurmator(void) {return urmator;}
    obiect_lista *redaprecedent(void) {return precedent;}
    void redainfo(char &c) { c = info;}
    void modific(char c) {info = c;}
    friend ostream &operator<<(ostream &flux, obiect_lista o)
    {
      flux << o.info << endl;
      return flux;
    }
    friend ostream &operator<<(ostream &flux, obiect_lista *o)
    {
      flux << o->info << endl;
```

```
    return flux;
  }
  friend istream &operator>>(istream &flux, obiect_lista &o)
  {
    cout << "Introduceti informatia: " << endl;
    flux >> o.info;
    return flux;
  }
};
```

Un lucru important de reținut în legătură cu definiția clasei *obiect_lista* este modul în care se definesc funcțiile sale *friend* (pe care le utilizează pentru a manipula ieșirea și intrarea din fluxuri), *inline* în cadrul definiției clasei, o construcție ușor diferită de cea pe care ați utilizat-o până acum. Ați fi putut la fel de simplu să declarați funcțiile *friend* în mod obișnuit. De exemplu, ați putea declara operatorul de inserare supraîncărcat, ca mai jos:

```
friend istream &operator>>(istream &flux, obiect_lista &o);
// Cod clasa
istream &operator>>(istream &flux, obiect_lista &o);
  {
    cout << "Introduceti informatia: " << endl;
    flux >> o.info;
    return flux;
  }
```

1180 FUNCȚIILE REDAURMATOR ȘI REDAPRECEDENT

C/C++

În definiția clasei prezentată în secțiunea 1179, există două definiții de funcții inline pentru funcțiile *redaprecedent* și *redaurmator*. La fel ca în cazul listei dublu înlănțuite construite anterior, utilizând structura *IntrareLista*, fiecare instanță a clasei *obiect_lista* păstrează doi pointeri: unul către elementul situat în listă înaintea obiectului și unul către elementul următor obiectului în listă. În loc de a apela funcții externe așa cum ați procedat în cazul listei construită pe baza structurii, lista construită pe baza unei clase vă permite să construiți funcții de deplasare chiar în cadrul clasei, cum arătăm în continuare:

```
obiect_lista *redaurmator(void) {return urmator;}
obiect_lista *redaprecedent(void) {return precedent;}
```

De aceea, atunci când programele dumneavoastră navighează prin listă, ele vor utiliza metodele *redaprecedent* și *redaurmator* pentru deplasarea de la un nod din listă către următorul și invers, ca mai jos:

```
nod = nod.redaurmator();
nod = nod.redaprecedent();
```

FUNCȚIILE DE SUPRAÎNCĂRCARE A OPERATORILOR

C/C++1181

Spre deosebire de clasa *Siruri*, care vă cerea să supraîncărcați majoritatea operatorilor relaționali și de calcul, lista dublu înlănțuită cere supraîncărcarea numai a operatorilor de intrare și ieșire. În cazul particular al listei înlănțuite pe care ați construit-o, operatorii de flux trebuie să fie capabili să controleze trei situații: un *obiect_lista*, un pointer către un *obiect_lista* și o intrare într-un *obiect_lista*. Prima și ultima situație sunt relativ ușor de înțeles și nu necesită explicații, în afară de aceea că ambele manipulează datele membre *info* (un membru de tip *char* în această situație). Totuși, este posibil să vă mire faptul că definiția clasei cere o a doua funcție supraîncărcată de extragere. După cum vă amintiți de când ați lucrat anterior cu o listă înlănțuită, deseori manipulați pointeri către un alt obiect din listă – astfel, crearea unei funcții de extragere capabilă să controleze un pointer către un obiect *obiect_lista* este utilă. Veți implementa trei funcții care supraîncarcă operatorii în cadrul programului dumneavoastră, ca mai jos:

```
friend ostream &operator<<(ostream &flux, obiect_lista *o)
   {
     flux << o.info << endl;
     return flux;
   }
friend ostream &operator<<(ostream &flux, obiect_lista *o)
   {
     flux << o->info << endl;
     return flux;
   }
friend istream &operator>>(istream &flux, obiect_lista &o)
   {
     cout << "Introduceti informatia: " << endl;
     flux >> o.info;
     return flux;
   }
```

Primele două funcții supraîncărcate afișează ceea ce programul a stocat anterior în cadrul membrului *info* al nodului. A treia funcție vă permite să introduceți informația în cadrul membrului *info*. Trebuie să acordați o atenție specială modului în care definiția clasei *obiect_lista* supraîncarcă operatorii flux – clasa *obiect_lista* îi definește ca funcții *friend* (pe care le utilizează pentru manipularea ieșirii și intrării din fluxuri) inline în cadrul definiției clasei. Definiția clasei *obiect_lista* utilizează o construcție ușor diferită decât cea utilizată anterior, dar ați fi putut la fel de bine să declarați funcțiile *friend* cum procedați de obicei. De exemplu, ați putea declara operatorul de inserare supraîncărcat, cum arătăm mai jos:

```
    friend istream &operator>>(istream &flux, obiect_lista &o);
 // Cod clasa
    istream &operator>>(istream &flux, obiect_lista &o);
      {
        cout << "Introduceti informatia: " << endl;
        flux >> o.info;
```

```
        return flux;
    }
```

1182 MOȘTENIREA CLASEI OBIECT_LISTA

C/C++

În secțiunile precedente ați definit și studiat clasa *obiect_lista* din care veți crea obiecte în cadrul unei liste dublu înlănțuite. Însă, trebuie să înțelegeți că această clasă definește numai informația despre fiecare obiect din cadrul listei – clasa *obiect_lista* nu furnizează programelor dumneavoastră informații despre lista însăși. În următoarea secțiune veți deriva clasa *lista_inl* din clasa *obiect_lista* pentru întreținerea informației despre listă. Înainte însă de a deriva clasa *lista_inl*, asigurați-vă că înțelegeți relațiile dintre fiecare obiect și listă, cum ilustrează figura 1182.

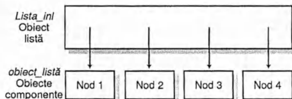


Figura 1182 *Relațiile dintre clasele* ***obiect_lista*** *și* ***lista_inl***.

1183 CLASA LISTELOR ÎNLĂNȚUITE

C/C++

După cum ați învățat în secțiunea 1182, clasa *obiect_lista* nu furnizează programelor dumneavoastră informații despre lista însăși, ci numai despre fiecare element al listei. Pentru a dispune de informația despre o listă dată de elemente, trebuie să derivați o a doua clasă, clasa *lista_inl*, din clasa *obiect_lista*. Clasa *lista_inl* trebuie să dețină doi pointeri, unul la începutul listei și altul la elementul final al listei. Ambii pointeri, de început și de final, sunt pointeri către obiecte *obiect_lista*. Constructorul *lista_inl* inițializează ambii pointeri la *NULL* de fiecare dată când programul dumneavoastră creează o nouă listă. Veți implementa întreaga clasă *lista_inl* ca mai jos:

```
class lista_inl : public obiect_lista {
   obiect_lista *start, *final;
public:
    lista_inl(void) {start = final = NULL;}
    void pastreaza(char c);
    void elimina(obiect_lista *ob);
    void lsinainte(void);
    void lsinapoi(void);
    obiect_lista *cauta(char c);
    obiect_lista *redastart(void) {return start;}
    obiect_lista *redafinal(void) {return final;}

 ;
```

Pe lângă pointerii *start* și *final*, clasa *lista_inl* mai implementează câteva funcții pe care programele dumneavoastră le pot folosi pentru a trata și a manipula lista. Funcțiile membre adiționale permit programelor dumneavoastră să efectueze următoarele acțiuni:

- Să plaseze un articol în listă
- Să înlăture un articol din listă
- Să afișeze lista în ordinea normală sau inversă
- Să găsească un anumit element din listă
- Să obțină pointerii către începutul și sfârșitul listei

Următoarele secțiuni examinează în detaliu implementarea fiecărei acțiuni din lista precedentă.

FUNCȚIA DE MEMORARE A CLASEI LISTELOR ÎNLĂNȚUITE

C/C++1184

După cum ați învățat, puteți folosi clase pentru a implementa mai ușor o listă dublu înlănțuită similară uneia create în secțiunile anterioare. După cum știți, una dintre cele mai importante acțiuni pe care programele dumneavoastră trebuie să le execute cu orice listă înlănțuită este inserarea de obiecte în listă. În clasa *lista_inl*, funcția membru *pastreaza* operează inserarea obiectelor în listă. În cadrul acestei clase, veți implementa funcția membru *pastreaza*, ca mai jos:

```
void lista_inl::pastreaza(char c)
 {
   obiect_lista *p;

   p = new obiect_lista;
   if (!p)
     {
       cout << "Eroare de alocare." << endl;
       exit(1);
     }
   p->info = c;
   if (start == NULL)
     {
       final = start = p;
     }
   else
     {
     p->precedent = final;
     final->urmator = p;
     final = p;
     }
}
```

Înainte ca programul dumneavoastră să poată insera un nou element (în acest caz unul de tip *char*) în listă, funcţia trebuie să creeze un nou obiect *obiect_lista* pentru a memora elementul. Funcţia *pastreaza* va încerca să creeze un nou obiect *obiect_lista* şi va ieşi din program dacă nu reuşeşte. Dacă reuşeşte, funcţia *pastreaza* va memora valoarea din parametrul său *c* în obiectul nou creat *obiect_lista*. Funcţia *pastreaza* va adăuga apoi noul obiect *obiect_lista* la sfârşitul listei. De asemenea, funcţia va actualiza corespunzător pointerii obiectului *lista_inl*, *start* şi *final*. Aşa cum aţi implementat-o în cadrul clasei *lista_inl*, funcţia *pastreaza* va adăuga întotdeauna noii membrii la sfârşitul listei. Însă, puteţi cu uşurinţă modifica funcţia sau clasa, astfel încât funcţia *pastreaza* să insereze noile obiecte la locaţia corectă din cadrul listei, pentru a crea o listă sortată, aşa cum aţi făcut cu alte liste înlănţuite din secţiunile precedente.

Aşa cum funcţia *pastreaza* vă arată cu claritate, *lista_inl* manevrează un set de obiecte de tipul *obiect_lista*. Tipul datelor pe care lista le păstrează este irelevant pentru clasa *lista_inl*. Cu alte cuvinte, trebuie să modificaţi numai clasa *obiect_lista* pentru a accepta păstrarea de date mai utile. După cum veţi învăţa în secţiunea 1191, clasa *lista_inl* se prezintă ca o construcţie generică pe care o veţi utiliza la crearea mai multor liste înlănţuite de mai multe tipuri, într-un singur program.

1185 FUNCŢIA DE ŞTERGERE A CLASEI LISTELOR ÎNLĂNŢUITE

C/C++

În secţiunea 1184 aţi adăugat funcţia *pastreaza* clasei *lista_inl*. Funcţia *pastreaza* vă permite să adăugaţi listei dumneavoastră noi obiecte de tip *obiect_lista*. Altă sarcină importantă, pe care gestionarul dumneavoastră de liste *lista_inl* trebuie să o efectueze, este eliminarea unor obiecte de tip *obiect_lista* din cadrul listei. Clasa *lista_inl* efectuează înlăturarea obiectelor cu ajutorul funcţiei membre *elimina* pe care o veţi implementa ca mai jos:

```
void lista_inl::elimina(obiect_lista *ob)
  {
    if(ob->precedent)
      {
        ob->precedent->urmator = ob->urmator;
        if(ob->urmator)
          ob->urmator->precedent = ob->precedent;
        else
          final = ob->precedent;
      }
    else
      {
        if(ob->urmator)
          {
          ob->urmator->precedent = NULL;
          start = ob->urmator;
          }
        else
          start = final = NULL;
      }
  }
```

Funcţia *elimina* şterge obiectul către care indică pointerul *ob* în cadrul listei. După cum aţi învăţat în exemplul de listă precedent, un obiect pe care trebuie să-l ştergeţi din listă se află într-unul din cele trei locuri posibile în cadrul listei: este fie primul element, fie ultimul element, fie un element din mijloc, ca în figura 1185.

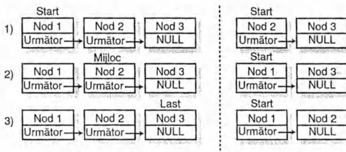


Figura 1185 *Cele trei cazuri posibile pe care le manipulează funcţia membru* ***elimina.***

Funcţia membru *elimina* controlează toate posibilităţile şi comprimă lista după ce a eliminat obiectul pe care nu-l mai doriţi.

FUNCŢIILE REDASTART ŞI REDAFINAL

C/C++1186

Singurele funcţii inline definite în clasa *lista_inl* sunt funcţiile *redastart* şi *redafinal*, arătate mai jos:

```
obiect_lista *redastart(void) {return start;}
obiect_lista *redafinal(void) {return final;}
```

Ambele funcţii *redastart* şi *redafinal* sunt funcţii de interfaţă. Puteţi apela funcţiile *redastart* şi *redafinal* din orice parte a programului pentru a deplasa pointerul listei către sfârşitul sau începutul listei. Alte funcţii membre din cadrul clasei pot, de asemenea, să apeleze *redastart* şi *redafinal* pentru a ajuta funcţiile să traverseze lista, aşa cum veţi învăţa în următoarea secţiune. Ambele funcţii *lsinapoi* şi *lsinainte* utilizează funcţiile *redastart* şi *redafinal* pentru a se poziţiona ele însele în cadrul listei, înainte de a genera ieşirea.

AFIŞAREA LISTEI ÎNLĂNŢUITE ÎN ORDINE ASCENDENTĂ

C/C++1187

O activitate comună pe care programele dumneavoastră o vor efectua cu orice listă înlănţuită este afişarea listei. În secţiunea 1188 veţi învăţa cum să afişaţi o listă începând de la ultimul element până la primul. Însă, programele dumneavoastră vor afişa frecvent o listă în ordinea în care programul a creat-o. Pentru a accepta afişarea ascendentă a unei liste, clasa *lista_inl* defineşte funcţia *lsinainte*, ca mai jos:

```
void lista_inl::lsinainte(void)
  {
    obiect_lista *temp;
```

```
    temp = redastart();
    do {
       cout << temp->info << " ";
       temp = temp->redaurmator();
       } while(temp);
    cout << endl;
  }
```

Funcția *lsinainte* nu acceptă parametri, pentru că întotdeauna ea va traversa întreaga listă. Atunci când apelați funcția *lsinainte* împreună cu un obiect de tipul *lista_inl*, ea utilizează mai întâi funcția *redastart* pentru a obține pointerul către primul obiect din listă. Apoi, funcția utilizează o buclă *do* și funcția *redaurmator* pentru a parcurge lista element cu element și a afișa datele conținute în fiecare element în ordine.

1188 AFIȘAREA LISTEI ÎNLĂNȚUITE ÎN ORDINE INVERSĂ

C/C++

În secțiunea 1187 ați creat funcția membru *lsinainte* pe care programele dumneavoastră o pot utiliza pentru a afișa o listă de tip *lista_inl* de la primul la ultimul element. După cum știți, programele dumneavoastră trebuie să afișeze frecvent o listă dublu înlănțuită în ordine inversă. Pentru a ajuta programele dumneavoastră să afișeze listele în ordine inversă, clasa *lista_inl* prevede funcția *lsinapoi*, arătată mai jos:

```
void lista_inl::lsinapoi(void)
  {
    obiect_lista *temp;
    temp = redafinal();
    do {
       cout << temp->info << " ";
       temp = temp->redaprecedent();
       } while(temp);
    cout << endl;
  }
```

Tot așa cum funcția *lsinainte* utiliza funcția *redastart* pentru a obține un pointer la primul membru al listei și apoi se deplasa înainte în listă, funcția *lsinapoi* utilizează funcția *redafinal* pentru a obține un pointer către ultimul element al listei. Apoi funcția utilizează o buclă *do* și funcția *redaprecedent* (care returnează un pointer către elementul precedent din listă) pentru a se deplasa înapoi în listă.

1189 CĂUTAREA ÎN LISTĂ

C/C++

După cum ați învățat în secțiunile precedente, obiectul *lista_inl* adaugă noile elemente la sfârșitul listei. Prin urmare, atunci când va trebui să căutați un element din listă, programul dumneavoastră va trebui să fie capabil să caute în listă acel element, pentru că altfel nu veți avea o înregistrare permanentă cu locul unde lista stochează obiectul. Pentru a vă ajuta să căutați într-o listă un anumit element, clasa *lista_inl* implementează funcția *cauta*, ca mai jos:

```
obiect_lista *lista_inl::cauta(char c)
  {
    obiect_lista *temp;
    temp = redastart();
    while(temp) {
      if(c == temp->info)
        return temp;
      temp = temp ->redaurmator();
    }
    return NULL;
  }
```

Funcția *cauta* începe procesarea apelând funcția *redastart*. După ce obține un pointer către primul obiect din listă, ea parcurge lista element cu element, încercând să găsească parametrul pe care l-a primit. Dacă funcția *cauta* a întâlnit elementul, ea returnează un pointer la acel element, dacă nu, ea returnează un pointer *NULL*.

IMPLEMENTAREA UNUI PROGRAM SIMPLU CARE FOLOSEȘTE CLASA LISTELOR ÎNLĂNȚUITE

C/C++1190

În secțiunile precedente ați creat clasele simple *obiect_lista* și *lista_inl*. Pentru a implementa aceste clase, programele dumneavoastră trebuie să creeze un obiect de tipul *lista_inl* și un pointer către un obiect de tipul *obiect_lista*. CD-ROM-ul care însoțește această carte cuprinde programul *use_link.cpp*, care implementează ambele clase. Următorul cod prezintă funcția *main* din programul *util_inl.cpp (use_link.cpp)*. Programul *util_inl.cpp* creează o listă simplă de trei elemente, apoi se deplasează înainte și înapoi în cadrul listei, adaugă și șterge elemente și se deplasează în listă „manual“ (utilizând *redastart* și *redaurmator*):

```
void main(void)
  {
    lista_inl lista;
    char c;
    obiect_lista *p;

    lista.pastreaza('1');
    lista.pastreaza('2');
    lista.pastreaza('3');

    cout << "Lista de la ultimul element la primul, apoi
      invers." << endl;
    lista.lsinapoi();
    lista.lsinainte();
    cout << endl;
    cout << "Parcurge 'manual' lista." << endl;
    p = lista.redastart();
    while(p) {
      p->redainfo(c);
```

```
    cout << c << " ";
    p = p->redaurmator();
  }
  cout << endl << endl;
  cout << "Cauta elementul 2." << endl;
  p = lista.cauta('2');
  if(p)
   {
     p->redainfo(c);
     cout << "A gasit: " << c << endl;
   }
  cout << endl;
  p->redainfo(c);
  cout << "Elimina elementul: " << c << endl;
  lista.elimina(p);
  cout << "Noua lista de la primul la ultimul element." << endl;
  lista.lsinainte();
  cout << endl;
  cout << "Adauga un element." << endl;
  lista.pastreaza('4');
  cout << "Lista de la primul la ultimul element." << endl;
  lista.lsinainte();
  cout << endl;
  p = lista.cauta('1');
  if(!p)
   {
     cout << "Eroare, elementul nu a fost gasit." << endl;
     return 1;
   }
  p->redainfo(c);
  cout << "Schimba " << c << " cu 5." << endl;
  p->schimba('5');
  cout << "Lista de la primul la ultimul element, apoi
    invers." << endl;
  lista.lsinainte();
  lista.lsinapoi();
  cout << endl;
  cout << "Introduceti informatia:" >> endl;
  cin >> *p;
  cout << p;
  cout << "Lista de la primul la ultimul element, din nou."
    << endl;
  lista.lsinainte();
  cout << endl;
  cout << "Lista dupa eliminarea primului element." << endl;
  p = lista.redastart();
  lista.elimina(p);
```

```
    lista.lsinainte();
    cout << endl;
    cout << "Lista dupa eliminarea ultimului element." << endl;
    p = lista.redafinal();
    lista.elimina(p);
    lista.lsinainte();
}
```

Când compilaţi şi executaţi programul, ecranul dumneavoastră va afişa următoarele:

```
Lista de la ultimul element la primul, apoi invers.
3 2 1
1 2 3
Parcurge 'manual' lista.
1 2 3
Cauta elementul 2.
A gasit: 2
Elimina elementul: 2
Noua lista de la primul la ultimul element.
1 3
Adauga un element.
Lista de la primul la ultimul element.
1 3 4
Schimba 1 cu 5
Lista de la primul la ultimul element, apoi invers.
5 3 4
4 3 5
Introduceti informatia:
1
Lista de la primul la ultimul element, din nou.
1 3 4
Lista dupa eliminarea primului element.
3 4
Lista dupa eliminarea ultimului element.
3
C:\>
```

CREAREA UNEI CLASE GENERICE LISTĂ DUBLU ÎNLĂNŢUITĂ

C/C++1191

În secţiunile precedente aţi creat clasele *obiect_lista* şi *lista_inl*, care acceptă un singur membru de tip *char* şi îl păstrează în interiorul listei. Însă, după cum ştiţi, o clasă de tip lista înlănţuită este mult mai utilă dacă acceptă informaţii de diferite tipuri şi dacă stochează informaţiile în cadrul listei. De exemplu, un program care utilizează trei liste diferite poate menţine o listă de tip *int*, o alta de tip *float* şi o a treia care păstrează tipul propriu (cum ar fi clasa *Carte* pe care aţi proiectat-o în secţiunile precedente). În loc de a crea o clasă *obiect_lista* şi una *lista_inl* pentru fiecare tip, trebuie să creaţi o clasă generică *obiect_lista* şi *lista_inl*. După cum aţi învăţat, atunci când creaţi o clasă generică, puteţi utiliza clasa cu

orice tip C++ sau personalizat. Clasa dumneavoastră generică sau clasa *șablon* (*template*), poate opera cu orice tip de date.

Un avantaj al creării unei clase generice este acela că ea separă *mecanismul* listei (deci, diferitele funcții pe care clasele *obiect_lista* și *lista_inl* le implementează) de datele pe care lista efectiv le stochează. Utilizarea unei clase generice pentru separarea mecanismului de date vă permite să creați mecanismul o dată, dar să îl utilizați de oricâte ori doriți.

1192 MEMBRII CLASEI GENERICE OBIECT_LISTA C/C++

După cum ați învățat în secțiunea 1191, o mai bună implementare a celor două clase ale dumneavoastră de tip listă dublu înlănțuită este aceea care utilizează definiții generice. Însă, pentru că ați derivat clasa *lista_inl* direct din clasa *obiect_lista*, trebuie să transformați mai întâi clasa *obiect_lista* în clasă generică, pentru a putea ulterior face același lucru cu clasa *lista_inl* (așa cum veți proceda în secțiunea 1193). Implementarea clasei generice *obiect_lista* este prezentată în continuare:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

template <class DataT> class obiect_lista {
  public:
    DataT info;
    obiect_lista<DataT> *urmator;
    obiect_lista<DataT> *precedent;
    obiect_lista(void)
      {
       info = 0;
       urmator = NULL;
       precedent = NULL;
      }
    obiect_lista(DataT c)
      {
       info = c;
       urmator = NULL;
       precedent = NULL;
      }
    obiect_lista<DataT> *redaurmator(void) {return urmator;}
    obiect_lista<DataT> *redaprecedent(void) {return precedent;}
    void redainfo(DataT &c) { c = info;}
    void schimba(DataT c) {info = c;}
    friend ostream &operator<<(ostream &flux, obiect_lista<DataT> o)
      {
       flux << o.info << endl;
       return flux;
      }
    friend ostream &operator<<(ostream &flux,
      obiect_lista<DataT> *o)
```

```
      {
        flux << o->info << endl;
        return flux;
      }
    friend istream &operator>>(istream &flux,
      obiect_lista<DataT> &o)
      {
        cout << "Introduceti informatia: " << endl;
        flux >> o.info;
        return flux;
      }
  };
```

După cum ați învățat în secțiunea 1120, următoarea declarație creează clasa generică *obiect_lista*:

```
template <class DataT> class obiect_lista{
```

Cuvântul cheie *template* atenționează compilatorul că urmează o descriere a unei clase generice. Operatorul *<class DataT>* informează compilatorul de faptul că clasa generică *obiect_lista* acceptă un singur tip generic. Atunci când declarați obiecte de tip *obiect_lista*, trebuie să indicați compilatorului ce tip de date vor utiliza respectivele instanțe, cum arătăm mai jos:

```
obiect_lista<char> lista_char;
obiect_lista<float> lista_float;
obiect_lista<personalizata> lista_personalizata;
```

Comenzile din fragmentul de cod precedent creează trei obiecte *obiect_lista*: unul de tip *char*, altul de tip *float* și un altul de tip *personalizat*. Date fiind următoarele definiții, obiectul dumneavoastră *obiect_lista* poate să păstreze efectiv membri de tip *DateCarte*:

```
class DateCarte {
  public:
    DateCarte(char *titlu, char *editura, char *autor);
    void arata_carte(void)
      {
        cout << "Carte: " << titlu << " de " << autor << endl
          << "Editura: " << editura << endl;
      };
  private:
    char titlu[64];
    char autor[64];
    char editura[64];
};
    // Cod suplimentar
void main(void)
  {
    obiect_lista<DateCarte> obiect1;
```

1193 CLASA GENERICĂ LISTA_INL



După cum ați învățat în secțiunea 1192, trebuie să precedați definiția unei clase generice cu cuvântul cheie *template* și cu un operator care corespunde tipului pe care compilatorul îl va implementa în locul tipului generic. Atunci când implementați clasa *lista_inl*, care derivă din clasa *obiect_lista*, trebuie, de semenea, să oferiți compilatorului informații despre tipul generic. Totuși, atunci când inițializați un obiect *lista_inl* cu un tip specificat, se vor inițializa automat toate obiectele *obiect_lista* utilizate de clasa *lista_inl* ca fiind de același tip specificat. Veți implementa definiția clasei generice *lista_inl*, ca mai jos:

```
template <class DataT> class lista_inl : public
  obiect_lista<DataT> {
    obiect_lista<DataT> *start, *final;
public:
    lista_inl() {start = final = NULL;}
    void pastreaza(DataT c);
    void elimina(obiect_lista<DataT> *ob);
    void lsinainte();
    void lsinapoi();
    obiect_lista<DataT> *cauta(DataT c);
    obiect_lista<DataT> *redastart() {return start;}
    obiect_lista<DataT> *redafinal() {return final;}
};
```

După cum puteți vedea, atât în cadrul clasei *lista_inl*, definite în această secțiune, cât și în cazul clasei *obiect_lista*, definită în secțiunea 1192, funcțiile și datele membre sunt definite în termenii obiectului generic *DataT*. În plus, funcțiile membre care primesc un obiect de un tip oarecare ca parametru (cum ar fi funcția *elimina* a clasei *lista_inl*) utilizează obiectul generic *DataT* împreună cu definiția parametrului.

1194 UTILIZAREA CLASELOR GENERICE CU O LISTĂ DE CARACTERE



În secțiunea 1190 ați creat programul *util_inl.cpp* care utiliza clasele inițiale *obiect_lista* și *lista_inl* pentru a manevra o listă scurtă de variabile de tip caracter. CD-ROM-ul însoțitor al acestei cărți cuprinde programul *use_glink.cpp* (util_ginl.cpp) care utilizează clasele generice de liste pentru a crea o listă de tip *char* identică listei create de programul *util_inl.cpp (use_link.cpp)*.

După cum veți vedea mai târziu, singura diferență semnificativă dintre programele *util_inl.cpp* și *util_ginl.cpp* este următoarea declarație:

```
lista_inl<char> lista;
char c;
obiect_lista<char> *p;
```

După cum știți, declarația *lista_inl<char> lista* creează obiectul *lista*. Obiectul *lista* acceptă membrul *info* de tipul *char*. Restul codului din funcția *main* este în esență identic, deoarece programul *util_ginl.cpp* știe că operează cu o listă de tip *char* ca și programul *util_inl.cpp*

din secțiunea 1190. În secțiunea 1195, însă, veți crea o listă de tipul *double* pentru a proba faptul că clasele generice lucrează cu mai multe tipuri.

UTILIZAREA CLASELOR GENERICE CU O LISTĂ DOUBLE

C/C++1195

În secțiunea 1194 ați utilizat clasele generice de liste pentru a crea o listă de tip *char*. Însă, după cum știți, un beneficiu remarcabil al claselor generice este că acestea permit programelor dumneavoastră să utilizeze aceeași definiție de bază pentru a crea mai multe tipuri de clase. CD-ROM-ul care însoțește această carte cuprinde programul *dbl_link.cpp (dbl_list.cpp)*, care utilizează clase generice pentru a stoca valori de tip *double*.

Programul *dbl_list.cpp* execută o prelucrare similară cu programul *util_ginl.cpp*, impementat de secțiunea 1194. Declarațiile, însă, sunt ușor diferite, ceea ce afectează întreaga acțiune a programului asupra listei, cum arătăm în continuare:

```
lista_inl<double> lista;
double c;
obiect_lista<double> *p;
```

Atunci când comenzile din cadrul funcției *main* manipulează *lista*, fiecare comandă operează cu valori *double*, în loc de valorile de tip *char*.

UTILIZAREA CLASELOR GENERICE CU O STRUCTURĂ

C/C++1196

În secțiunile precedente ați creat o listă dublu înlănțuită simplă cu clasele dumneavoastră generice și tipuri simple de date. Veți descoperi că este necesar să lucrați intens cu șabloanele generice înainte de a le putea utiliza cu structuri mai complexe. Pentru a înțelege de ce funcții care operează pentru tipuri simple sunt insuficiente pentru structurile mai complexe, analizați următoarea funcție din definiția generică creată anterior:

```
friend ostream &operator<<(ostream &flux, obiect_lista<DataT> o)
  {
    flux << o.info << endl;
    return flux;
  }
```

Funcția de supraîncărcare a operatorului de inserție supraîncarcă fluxul de ieșire și plasează informația stocată de *info* în cadrul fluxului. Atunci când *info* este un tip simplu, această comandă este suficientă. Să analizăm, însă, următoarea definiție de clasă:

```
class Carte {
  public:
    Carte(char *titlu, char *autor, char *editura, float pret);
      // Constructor
    Carte(void) {};
    void arata_titlu(void);
    float da_pret(void);
    void arata(void);
```

```
        void atrib_editura(char *nume);
        bool operator==(Carte op2)
      private:
        char titlu[256];
        char autor[64];
        float pret;
        char editura[256];
        void arata_editura(void);
    };
```

După cum știți, nu puteți să inserați pur și simplu obiecte de un tip mai complex (cum ar fi tipul *Carte*) în cadrul fluxului de ieșire. Trebuie să inserați membrii individuali în fluxul de date. Însă, deoarece tipul generic consideră toate obiectele din această clasă ca membri *info* ai clasei *obiect_listă*, nu puteți să supraîncărcați operatorul supraîncărcat al clasei *obiect_lista* pentru *ostream*. Programele dumneavoastră trebuie să utilizeze identificarea tipului în timpul rulării pentru a determina ce tip utilizează lista curentă și pentru a apela funcția de supraîncărcare corespunzătoare, în mod explicit, pentru a afișa corect datele. Altfel, puteți forța toate clasele care utilizează lista să accepte funcția *arata* sau alte funcții standard, pe care programele ar putea să le utilizeze ulterior pentru a afișa informația din cadrul obiectului. Secțiunea 1199 prezintă un program care utilizează o funcție standard *arata* pentru a afișa informații complexe.

1197 SUPRAÎNCĂRCAREA OPERATORULUI DE COMPARARE ==

C/C++

După cum ați învățat în secțiunea 1196, dacă utilizați clasele *obiect_lista* și *lista_inl* cu tipuri de date mai complexe, trebuie ca mai întâi să modificați câteva dintre definițiile generale din cadrul claselor generice. În plus, clasele componente (pe care le veți stoca în cadrul listei) trebuie să supraîncarce operatorul de comparare ==. Deoarece metoda *cauta* examinează un obiect și compară valoarea sa *info* cu fiecare valoare *info* din listă, precum și datorită faptului că *info* se referă la un obiect clasă, nu puteți utiliza un cod cum ar fi cel de mai jos în cadrul clasei generice:

```
template <class DataT> obiect_lista<DataT>
*lista_inl<DataT>::cauta(obiect_lista<DataT> ob)
  {
    obiect_lista<DataT> *temp;

    temp = start;
    while(temp) {
      if(ob.info==temp->info)
        return temp;
      temp = temp->redaurmator();
    }
    return NULL;
  }
```

Dacă încercați să utilizați codul ca în exemplul precedent, compilatorul va genera o eroare, pentru că el știe că nu puteți compara explicit două obiecte așa cum comparați două tipuri

simple de date. De aceea, când listele dumneavoastră generice utilizează tipuri complexe de date, trebuie să vă asigurați că tipurile dumneavoastră complexe de date supraîncarcă operatorul ==. De exemplu, următorul fragment de cod supraîncarcă operatorul == pentru clasa *Carte*, utilizată în multe programe precedente:

```
bool Carte::operator==(Carte ob2)
  {
    if(titlu != op2.titlu)
      return false;
    if(autor != op2.autor)
      return false;
    if(editura != op2.editura)
      return false;
    if(pret != op2.pret)
      return true;
  }
```

Funcția de supraîncărcare a operatorului == analizează toți membrii din cadrul obiectului *Carte*. Dacă vreunul dintre membri diferă de cel comparat, comparația va returna *false* și funcția se încheie. Dacă toți membrii sunt identici, funcția va returna *true*. Observați că această implementare particulară utilizează date de tip *bool*, dar ar putea la fel de bine să utilizeze date de tip *int*.

ALTE PERFECȚIONĂRI ADUSE LISTEI GENERICE

C/C++1198

Pe măsură ce programele dumneavoastră lucrează mai mult cu funcții de listă generică, veți observa, probabil, că există câteva implementări pe care puteți fie să le adăugați la listă, fie să le modificați în cadrul definiției de bază a listei. Așa cum au identificat secțiunile precedente, o limitare semnificativă a clasei *lista_inl* este aceea că nu își poare sorta elementele înaintea inserării lor în cadrul listei. Este posibilă modificarea funcției *pastreaza* de bază sau crearea unei funcții *pastreaza_sort*. Este posibilă crearea unui șablon de clasă generică sortată.

În plus, este posibil ca clasa dumneavoastră de listă (dacă nu face automat sortarea elementelor) să fie capabilă de a stoca informații în mai multe locuri din listă. De exemplu, puteți adăuga funcțiile membre *pastreaza_start, pastreaza_final* și *cauta_pastreaza*. (Funcția membru *cauta_pastreaza* caută un element specificat în listă și păstrează noua informație înainte sau după elementul căutat.) Astfel, este posibil să adăugați un parametru de tip întreg la funcția membru *pastreaza*, care vă permite să invocați *pastreaza* cu implementări diferite. În final, puteți modifica *pastreaza* în așa fel încât să primească o instanță obiect a datelor, și nu un tip simplu de date și să adauge obiectul la listă. Secțiunea 1199 arată modul în care puteți implementa funcția *pastreaza* care primește datele obiectului.

UTILIZAREA OBIECTELOR CU FUNCȚIA DE MEMORARE

C/C++1199

După cum ați învățat, clasele generice de liste înlănțuite pe care le-ați creat, deși sunt de bază și utile pentru controlul tipurilor simple de date, au anumite probleme atunci când încercați

să păstrați tipuri de date complexe în cadrul listei. Una dintre cele mai semnificative probleme care apar cu această construcție curentă este modul de inserare a informațiilor într-un nou *obiect_lista*. De exemplu, bazându-vă pe modalitatea în care ați scris funcția inițială *pastreaza* și pe ceea ce știți destre funcțiile constructor, puteți încerca să scrieți adăugarea de noi elemente listei de tip *Carte*, cum arătăm mai jos:

```
lista.pastreaza("Jamsa's C/C++ Programmer's Bible",
            "Jamsa & Klander", "Jamsa Press", 49.95);
```

Din păcate, compilatorul nu va recunoaște existența funcției constructor și va returna o eroare dacă încercați să creați elementele noi ale listei în maniera prezentată în fragmentul de cod precedent. În plus, codul mai mult vă încurcă decât vă folosește. Următorul cod, însă, este mai clar – în special dacă programul dumneavoastră prelucrează informația *Carte* înainte de a încerca să stocheze informația în listă:

```
Carte cbib("Jamsa's C/C++ Programmer's Bible",
        "Jamsa & Klander", "Jamsa Press", 49.95);
// Cod program
lista.pastreaza(cbib);
```

În cel de al doilea fragment de cod, creați mai întâi obiectul, apoi transmiteți obiectul către funcția *pastreaza*. Datorită modului în care ați proiectat structura generică, aceasta va prelucra complet obiectul *cbib* și îl va adăuga listei. CD-ROM-ul care însoțește această carte cuprinde programul *bk_list.cpp (list_crt.cpp)* care adaugă trei obiecte listei în modalitatea prezentată de cel de al doilea fragment de cod utilizat de această secțiune. Datorită necesității de a modifica modul în care clasa operează ieșirea, programul nu efectuează toate acțiunile de ieșire pe care programele anterioare le-au efectuat. Totuși, programul *list_crt.cpp* parcurge lista element cu element și generează o ieșire. Atunci când compilați și executați programul *list_crt.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Iata cateva elemente.
Parcurge 'manual' lista.
Titlu: Jamsa's C/C++ Programmer's Bible
Editura: Jamsa Press
Titlu: 1001 Visual Basic Programmer's Tips
Editura: Jamsa Press
Titlu: Hacker Proof
Editura: Jamsa Press
C:\>
```

1200 SCRIEREA UNEI FUNCȚII PENTRU A DETERMINA LUNGIMEA LISTEI

C/C++

În secțiunea 1198 ați învățat despre perfecționările pe care programele dumneavoastră pot să le aducă claselor generice de liste. O perfecționare abordată de secțiunea 1198 este adăugarea unei funcții care traversează lista și returnează numărul total de elemente din cadrul listei. Următorul fragment de cod furnizează un exemplu de implementare a funcției membru *lunglist*, care numără elementele:

```
template <class DataT> int lista_inl<DataT>::lunglist(void)
{
   obiect_lista<DataT> *temp;
   int numar = 0;
   temp = start;
   do {
      temp = temp->redaurmator();
      numar = numar + 1;
     } while(temp);
   cout << "Numarul de elemente ale listei: " << cout << endl;
   return numar;
}
```

După cum puteți vedea, funcția membru *lunglist* efectuează aceeași prelucrare ca funcția *lsinainte*, cu excepția că reține un număr, pe măsură ce traversează lista. Funcția membru *lunglist* afișează apoi numărul la încheierea prelucrării și returnează numărul de elemente către valoarea de tip *lvalue* în cadrul instrucțiunii apelante. CD-ROM-ul care însoțește această carte cuprinde programul *cnt_lst.cpp (nr_lista.cpp)* care efectuează aceeași prelucrare ca programul *use_glink.cpp (util_ginl.cpp)* prezentat în secțiunea 1194. Însă, programul *nr_list.cpp* invocă și funcția *lunglist (list length)* în diferite puncte din cadrul execuției programului.

PREZENTAREA BIBLIOTECII DE ȘABLOANE STANDARD

C/C++ 1201

Biblioteca standard de șabloane (Standard Template Library) sau STL este o bibliotecă C++ de clase container (cum ar fi listele înlănțuite), algoritmi și iteratori; aceasta dispune de mulți algoritmi de bază și structuri de date ale informaticii (cum ar fi sortarea, maparea și funcții matematice). Biblioteca standard de șabloane este o bibliotecă generică, ceea ce înseamnă că cei care au creat-o au parametrizat intens componentele bibliotecii: aproape fiecare componentă a bibliotecii standard de șabloane este un șablon. Trebuie să vă asigurați că înțelegeți modul în care lucrează șabloanele în C++ înainte de a utiliza biblioteca standard de șabloane. Puteți utiliza biblioteca standard de șabloane pentru a realiza repede și ușor următoarele:

- Crearea listelor sortate de obiecte
- Crearea listelor sortate de obiecte ce dețin aceeași cheie
- Manipularea structurilor complexe de date într-o manieră simplă și directă
- Efectuarea de manevre complexe asupra informației depozitate într-un container cu ajutorul algoritmilor predefiniți.

În următoarele cincizeci de secțiuni veți utiliza componentele bibliotecii standard de șabloane pentru a scrie câteva programe. Înainte de a începe, ar trebui să rețineți următoarele idei importante:

1. Deoarece compilatorul *Turbo C++ Lite* nu acceptă definițiile generice, nu veți putea utiliza biblioteca standard de șabloane cu acest compilator.

2. Atât *Visual C++* cât și *Borland C++ 5.02 for Windows* cuprind fișierele antet pentru aproape toată biblioteca standard de șabloane, ceea ce înseamnă că puteți utiliza multe dintre facilitățile bibliotecii standard de șabloane cu aceste compilatoare, fără modificări ulterioare.
3. Atunci când compilatorul dumneavoastră nu conține biblioteca standard de șabloane, puteți descărca fișierele antet necesare prin Internet. Pentru a descărca fișierele bibliotecii standard de șabloane, vizitați situl Web al firmei Silicon Graphics la *http://www.sgi.com/Technology/STL/index.html*, după cum arată figura 1201.

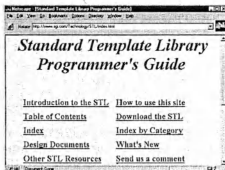


Figura 1201 *Pagina Silicon Graphics pentru STL.*

1202 *FIȘIERELE ANTET ALE BIBLIOTECII DE ȘABLOANE STANDARD*



După cum ați învățat în secțiunile precedente, biblioteca standard de șabloane pune la dispoziție clase și funcții generice pe care programele dumneavoastră le pot utiliza pentru a crea facilități suplimentare în limbajul C++. Din cauza numărului mare de clase coținute de biblioteca standard de șabloane, proiectanții bibliotecii au împărțit biblioteca în mai multe fișiere antet pentru a reduce durata compilării. Tabelul 1202 detaliază fișierele antet.

Nume fișier	Descriere
algo.h	Curpinde toți algoritmii din biblioteca standard de șabloane. Următoarele secțiuni vor detalia acești algoritmi (numiți, de asemenea și *algorith.h* în unele implementări de biblioteci standard).
bool.h	Definește datele de tip *bool*.
bvector.h	Definește obiectele *bit_vector* pe care programele dumneavoastră le pot utiliza pentru a întreține structurile de biți de tip matrice.
deque.h	Definește obiectele *deque* pe care programele dumneavoastră le pot utiliza pentru a crea structuri de tip matrice cu care puteți opera atât de la începutul, cât și de la sfârșitul lor.
function.h	Cuprinde operatori, obiecte funcții și adaptori de funcții care controlează clasele din biblioteca standard

Nume fișier	Descriere
iterator.h	Definește etichete (*tag*) pentru iteratori, iteratori de flux și adaptori de iteratori pentru clasele bibliotecii standard de șabloane.
list.h	Definește obiectele generice cu liste înlănțuite.
map.h	Definește clasa *map*, o listă dublu înlănțuită, sortată cu o valoare cheie și o valoare de date.
multimap.h	Definește clasa *multimap*, o listă înlănțuită cu una sau mai multe valori cheie și o valoare de date. Obiectele *multimap* acceptă unul sau mai multe câmpuri sortate în cadrul listei.
multiset.h	Definește clasa *multiset* care premite programelor dumneavoastră să reprezinte liste sortate într-o modalitate care permite căutarea, inserarea și îndepărtarea unui element oarecare cu un număr de operații proporțional cu logaritmul numărului de elemente din secvență (o valoare numită *timp logaritmic*). Spre deosebire de clasa *set*, clasa *multiset* poate utiliza mai multe chei de sortare.
pair.h	Definește clasa *pair* care premite programelor dumneavoastră să stocheze două valori (de același tip sau de tipuri diferite) în cadrul unui singur obiect.
random.c	Definește un generator de numere aleatoare. Puteți include fișierul *random.c* dacă programele dumneavoastră utilizează algoritmul *random_shuffle*.
set.h	Definește clasa *set* care permite programelor dumneavoastră să reprezinte liste înlănțuite sortate într-o modalitate care permite căutarea, inserția și îndepărtarea unui element oarecare în timp logaritmic. Spre deosebire de clasa *multiset*, clasa *set* trebuie să utilizeze o singură cheie de sortare.
stack.h	Definește obiectele stivă pe care programele dumneavoastră le pot utiliza pentru a controla secvențe de elemente de lungimi variate. Obiectul alocă și eliberează spațiul de depozitare pentru secvența controlată.
tempbuf.c	Un fișier program care introduce un buffer auxiliar pentru algortimii *get_temorary_buffer*, *stable_partition*, *inplace_merge* și *stable_sort*. Fișierul antet *algo.h* include automat atât *tempbuf.c*, cât și *tempbuf.h*. Ar fi bine să nu includeți niciodată, în mod direct, acest fișier în cadrul programelor dumneavoastră.
Tempbuf.h	Cuprinde prototipuri de funcții și definiții de clase pentru fișierul program *tempbuf.c*.
vector.h	Definește clasa șablon *vector*, un obiect care controlează secvențe de elemente de diferite lungimi. Spre deosebire de clasele *list*, *map* și *set*, clasa *vector* este o listă simplu înlănțuită pe care compilatorul o tratează ca pe o matrice.

***Tabelul 1202** Fișierele antet pentru biblioteca standard de șabloane.*

***Observație:** Există și alte fișiere antet ale bibliotecii standard de șabloane pe lângă acestea, dar ele sunt utilizate în mod special pentru anumite compilatoare de C++ sub DOS/Windows care nu acceptă automat mai multe modele de memorie. Nu veți avea nevoie de aceste fișiere antet dacă utilizați **Visual C++** sau **C++ 5.02 for Windows** al firmei Borland. Însă, dacă utilizați un alt compilator, consultați documentația compilatorului pentru a determina dacă acesta solicită alte fișiere antet.*

1203 CONTAINERELE



Unul dintre blocurile constitutive fundamentale ale bibliotecii standard de şabloane este *containerul*. Un container este un obiect care stochează colecţii de alte obiecte (de aceleaşi tipuri). De exemplu, clasa *lista_inl* pe care aţi creat-o în secţiunea 1190 este un container pentru obiecte ale clasei *obiect_lista*. Figura 1203.1 descrie un model de container şi obiectele pe care containerul le păstrează în interiorul său.

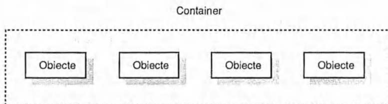


Figura 1203.1 *Modelul logic de container şi obiectele conţinute de el.*

Pe măsură ce continuaţi să scrieţi programe din ce în ce mai complexe, veţi descoperi că utilizarea containerelor (fie cele din biblioteca standard de şabloane, fie din alte biblioteci sau proiectate chiar de dumneavoastră) devine din ce în ce mai importantă pentru programele dumneavoastră. În loc să încercaţi să manipulaţi un număr mare de obiecte individuale, ceea ce cere un număr similar de variabile, bucle, teste şi aşa mai departe, puteţi să introduceţi mai multe obiecte într-un singur container, ceea ce face mai simplă manipularea obiectelor. De exemplu, este mult mai uşor să lucrezi cu o matrice de 10 întregi decât cu 10 variabile întregi. De fapt, multe dintre programele pe care le-aţi scris până acum utilizează cel mai simplu tip de container: o matrice. Figura 1203.2 descrie o matrice ca pe un container.

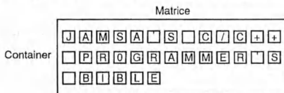


Figura 1203.2 *Matricea ca un container.*

În secţiunile următoare veţi învăţa mai mult despre containerele bibliotecii standard de şabloane şi le veţi implementa. În secţiunea 1204, însă, veţi utiliza clasele *lista_inl* şi *obiect_lista* pentru a recapitula conceptul de container.

1204 UTILIZAREA UNUI EXEMPLU DE CONTAINER



În secţiunea 1203 aţi învăţat despre containere şi despre modul în care programele dumneavoastră vor folosi clasele container pentru a întreţine informaţia despre grupuri de

obiecte. După cum aţi învăţat în secţiunea 1201, biblioteca standard de şabloane cuprinde diferite tipuri de containere (inclusiv *list, set, map, deque*). În secţiunea 1205 veţi învăţa noţiunile de bază despre tipurile bibliotecii standard de şabloane.

Însă, înainte de a continua să învăţaţi despre biblioteca standard de şabloane, este important să analizaţi o clasă generică pe ace aţi creat-o anterior, pentru a determina dacă are caracteristicile de container. În secţiunile anterioare aţi proiectat şi implementat clasa *lista_inl*, pe care aţi definit-o generic aşa cum arătăm mai jos:

```
template <class DataT> class lista_inl : public obiect_lista<DataT>
  {
     obiect_lista<DataT> *start, *final;
   public:
     lista_inl(void) {start = final = NULL;}
     void pastreaza(DataT c);
     void elimina(obiect_lista<DataT> *ob);
     void lsinaite(void);
     void lsinapoi(void);
     obiect_lista<DataT> *cauta(DataT c);
     obiect_lista<DataT> *redastart(void) {return start;}
     obiect_lista<DataT> *redafinal(void) {return final;}
  };
```

Clasa *lista_inl* este, desigur, derivată din clasa *obiect_lista*, pe care aţi definit-o ca mai jos (pentru claritate, această reluare relocalizează funcţiile *friend* inline în afara definiţiei clasei):

```
template <class DataT> class obiect_lista
  {
   public:
     DataT info;
     obiect_lista<DataT> *urmator;
     obiect_lista<DataT> *precedent;
     obiect_lista(void)
      {
        info = 0;
        urmator = NULL;
        precedent = NULL;
      }
     obiect_lista(DataT c)
      {
        info = c;
        urmator = NULL;
        precedent = NULL;
      }
     obiect_lista<DataT> *redaurmator(void) {return urmator;}
     obiect_lista<DataT> *redaprecedent(void) {return precedent;}
     void redainfo(DataT &c) {c = info;}
     void schimba(DataT c) {info = c;}
     friend ostream &operator<<(ostream &flux,
```

```
      obiect_lista<DataT> o)
    friend ostream &operator<<(ostream &flux,
      obiect_lista<DataT> *o)
    friend istream &operator>>(istream &flux,
      obiect_lista<DataT> &o)
  };
```

Deoarece clasa *obiect_lista* este generică, ea va accepta date de orice tip. Fiecare dintre următoarele declaraţii este validă:

```
    obiect_lista<int> obiect_int;
    obiect_lista<float> obiect_float;
    obiect_lista<char> obiect_char;
    obiect_lista<Carte> obiect_carte;
```

Definiţia generică a clasei înseamnă că tipurile ei nu au importanţă (cu alte cuvinte, programele dumneavoastră pot la fel de bine să definească *lista int* sau *float* şi funcţiile membre ale clasei *lista* vor lucra la fel cu oricare din ele). După cum ştiţi, un container operează cu obiecte individuale de tip cunoscut sau necunoscut fără a se ocupa de tipul obiectelor. De exemplu, fiecare dintre următoarele declaraţii de matrice este validă:

```
int matriceintregi[10];
float matricefloat[10];
char matricechar[10];
Carte matricecarte[10];
```

Deşi informaţia pe care o păstrează fiecare element din cadrul matricelor este diferită, matricele sunt identice. Deci, fiecare matrice conţine zece elemente de acelaşi tip, iar utilizatorul poate parcurge indicii matricei pentru a le accesa.

Clasa *lista_inl* este similară unei matrice. Datorită faptului că ea este derivată din clasa *obiect_lista*, compilatorul include explicit într-un tip fiecare instanţă a listei (cu alte cuvinte, dacă toate obiectele *obiect_lista* sunt *double*, obiectul *lista_inl* trebuie să fie de asemenea de tip *double*). Însă, clasa *lista_inl* însăşi nu se preocupă de natura obiectelor pe care le păstrează; ea păstrează acele obiecte şi vă pune la dispoziţie modalităţi de navigare prin *lista_inl*. Cu alte cuvinte, clasa *lista_inl* este în mod clar un container.

1205 PREZENTAREA CONTAINERELOR BIBLIOTECII STANDARD DE ŞABLOANE



După cum aţi învăţat, un container este unul dintre blocurile constitutive fundamentale ale bibliotecii standard de şabloane. În secţiunea 1204, aţi analizat modul în care aţi proiectat şi implementat anterior mai multe containere în programele dumneavoastră, chiar dacă nu vă dădeaţi seama că sunt containere. După cum veţi învăţa, biblioteca standard de şabloane defineşte implementări similare sau corelate cu multe dintre containerele pe care le-aţi utilizat anterior, precum şi alte containere pe care probabil că nu le-aţi întâlnit. Biblioteca standard de şabloane acceptă două tipuri de bază de containere: *containere secvenţiale* şi *containere asociative*. *Containerele secvenţiale* sunt obiecte care conţin colecţii într-o aranjare strict liniară. Biblioteca standard de şabloane acceptă următoarele trei *containere secvenţiale*:

- *vector<T>*: clasa *vector<T>* oferă acces aleator, de tipul matricelor, la o secvenţă de obiecte. În decursul execuţiei programului dumneavoastră, lungimea obiectului *vector* poate varia. Programele dumneavoastră pot efectua inserări şi ştergeri la capătul secvenţei. În general, veţi utiliza vectorii pentru a menţine informaţii nesortate într-o serie.
- *deque<T>*: Clasa *deque<T>* oferă acces aleator la o secvenţă de obiecte. Ca şi în cazul *vectorilor*, pe parcursul execuţiei programului, lungimea obiectului *deque* poate varia. Programele dumneavoastră pot efectua inserări şi ştergeri atât la începutul, cât şi la sfârşitul secvenţei. Veţi utiliza în general clasa *deque* pentru a păstra informaţii nesortate în serii pentru care nu sunteţi sigur la care capăt îi veţi adăuga informaţii.
- *list<T>*: clasa *list<T>* oferă acces la o secvenţă de obiecte. Ca şi în cazul claselor *vector* şi *deque*, când programul dumneavoastră se execută, lungimea obiectului *list* poate varia. Programele dumneavoastră pot efectua inserări şi ştergeri oriunde în secvenţă. Veţi utiliza listele în general pentru a manevra informaţii nesortate în serii în care nu sunteţi sigur la ce poziţie veţi insera informaţii.

Pe de altă parte, containerele asociative oferă programelor dumneavoastră o modalitate simplă de a regăsi rapid obiecte din colecţia conţinută de clasa container. Containerele asociative utilizează chei pentru a realiza regăsirea rapidă a obiectelor. Dimensiunea colecţiei poate varia în decursul execuţiei (aşa cum pot face toate containerele bibliotecii standard de şabloane, ceea ce le dă denumirea de containere *dinamice*, pe când matricele şi alte containere simple sunt containere *statice*). Un container asociativ menţine colecţia în ordine, bazându-se pe o funcţie de comparaţie, obiect al clasei *Compare*. Biblioteca standard de şabloane acceptă următoarele patru containere asociative:

- *set<T, Compare>*: clasa *set* acceptă chei unice (ceea ce înseamnă că obiectele clasei conţin cel mult una din fiecare valoare a cheii) şi oferă posibilitatea regăsirii rapide a cheii. Veţi utiliza în general clasa *set* pentru a manevra informaţii sortate ce utilizează o singură cheie de sortare, cum ar fi o mulţime simplă de numere întregi.
- *multiset<T, Compare>*: Clasa *multiset* acceptă chei duplicate (ceea ce înseamnă că obiectele clasei pot să conţină mai multe copii ale aceleiaşi valori cheie) şi oferă posibilitatea regăsirii rapide a cheilor. În general veţi utiliza clasele *multiset* pentru a manevra informaţii sortate care utilizează mai multe chei de sortare, care pot fi chiar data însăşi, cum ar fi o serie sortată de coordonate ale unei grile.
- *map<Key, T, Compare>*: Clasa *map* acceptă chei unice (ceea ce înseamnă că obiectele clasei conţin cel mult una din fiecare valoare cheie) şi oferă posibilitatea regăsirii rapide a altui tip *T* pe baza cheilor. Veţi utiliza în general clasele *map* pentru a manevra informaţii ordonate ce utilizează doar o cheie de sortare, cum ar fi o bază de date simplă cu numere de telefon.
- *multimap<Key, T, Compare>*: clasa *multimap* acceptă chei duplicate (ceea ce înseamnă că obiectele clasei pot să conţină mai multe copii ale aceleiaşi valori cheie) şi oferă posibilitatea regăsirii rapide a altui tip *T* pe baza cheilor. Veţi utiliza în general clasele *multimap* pentru a manevra informaţii ordonate ce utilizează chei multiple de sortare, cum ar fi o bază de date complexă a clienţilor.

1206 CONTAINERELE DE AVANSARE ȘI CONTAINERELE REVERSIBILE

C/C++

După cum ați învățat în secțiunea 1205, programele dumneavoastră care utilizează biblioteca standard de șabloane pot accesa două tipuri de bază de containere: containerele secvențiale și containerele asociative. Biblioteca standard de șabloane derivă ambele tipuri specificate de containere din tipurile mai generale de containere: *containerele de avansare* și *containerele reversibile*. În esență, considerați containerele de avansare ca o listă simplu înlănțuită. Containerele secvențiale derivă din containerele de avansare. Programele dumneavoastră pot lucra cu containerele de avansare numai într-o singură direcție. Figura 1206.1 arată modul în care programele dumneavoastră pot accesa containerele de avansare.

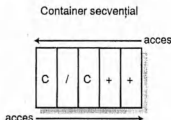


Figura 1206.1 *Programele dumneavoastră pot accesa containerele de avansare numai într-o singură direcție.*

Containerele reversibile, pe de altă parte, derivă din listele dublu înlănțuite. Containerele reversibile pun la dispoziție mijloace simple pentru ca programul dumneavoastră să poată parcurge o listă înainte și înapoi, pentru a ajunge mai ușor la începutul sau la finalul listei și așa mai departe. Programele dumneavoastră vor crea containerele reversibile în mod dinamic. Biblioteca standard de șabloane derivă containerele asociative din ambele tipuri de containere: cele de avansare și cele reversibile. Figura 1206.2 ilustrează modul în care programele dumneavoastră pot accesa containerele asociative.

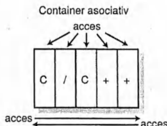


Figura 1206.2 *Programul dumneavoastră poate accesa Containerele Asociative în orice punct al containerului.*

CONTAINERELE SECVENȚIALE ALE BIBLIOTECII STANDARD DE ȘABLOANE

C/C++1207

După cum ați învățat în secțiunea 1205, un container secvențial este un obiect care stochează colecții de alte obiecte într-o ordine strict lineară. Biblioteca standard de șabloane acceptă trei containere secvențiale: *vectori*, *deque* și *liste*, precum și două containere derivate, *bit_vector* și *slist*. Pentru a înțelege modul în care programele dumneavoastră vor utiliza containerele secvențiale, să analizăm următorul program, *vector1.cpp*, care utilizează fișierul antet *vector.h* pentru a crea un vector gol de întregi, apoi lucrează cu acest vector:

```
#include <iostream.h>
#include <vector.h>

using namespace std;
typedef vector<int> INTVECTOR;
const ARRAY_SIZE = 4;

void main(void)
 {
   // vectorul alocat in mod dinamic nu contine la inceput
   // elemente.
   INTVECTOR Vectorul;

   // Intializeaza vectorul pentru a contine membrii
   // [100, 200, 300, 400]
   for (int nFiecareElem = 0; nFiecareElem < ARRAY_SIZE;
     nFiecareElem++)
    Vectorul.push_back((nFiecareElem + 1) * 100);

   cout << "Primul element: " << Vectorul.front() << endl;
   cout << "Ultimul element: " << Vectorul.back() << endl;
   cout << "Elementele din vector: " << Vectorul.size() << endl;

   // Sterge ultimul element al vectorului. Retineti ca vectorul
   // este cu baza 0, astfel incit Vectorul.end() indica de fapt
   // un element dincolo de capat.
   cout << "Stergem ultimul element." << endl;
   Vectorul.erase(Vectorul.end() - 1);
   cout << "Noul ultim element este: " << Vectorul.back() << endl;

   // Sterge primul element al vectorului.
   cout << "Stergem primul element." << endl;
   Vectorul.erase(Vectorul.begin());
   cout << "Noul prim element este: " << Vectorul.front() << endl;
   cout << "Elementele din vector: " << Vectorul.size() << endl;
 }
```

Programul *vector1.cpp* declară un vector gol de întregi, apoi inițializează vectorul cu membrii [100, 200, 300 și 400]. Apoi programul utilizează funcția membră *vector.front* pentru a obține și a afișa primul element al vectorului. După ce afișează primul element al vectorului, programul utilizează funcția membru *vector.back* pentru a obține și afișa ultimul element al vectorului. De asemenea, programul utilizează funcția membru *vector.size* pentru

a afișa numărul de elemente ale vectorului. După ce afișează numărul elementelor din vector, programul utilizează funcția membru *vector.end* pentru a se poziționa dincolo de capătul vectorului, scade unu din această poziție și utilizează funcția membru *vector.erase* pentru a înlătura ultimul element al vectorului. După ce șterge ultimul element, programul utilizează funcția membru *vector.back* pentru a afișa noul element de pe ultima poziție. După ce afișează noul ultim element, programul utilizează funcția membru *vector.erase* împreună cu funcția membru *vector.begin* pentru a șterge primul element al vectorului și apoi utilizează funcția membru *vector.front* pentru a afișa noul prim element. În final, programul utilizează funcția membru *vector.size* pentru a afișa numărul de elemente rămase în vector. Atunci când compilați și executați programul *vector1.cpp*, pe ecranul dumneavoastră vor apărea următoarele:

```
Primul element: 100
Ultimul element: 400
Elementele din vector: 4
Stergem ultimul element.
Noul ultim element este: 300
Stergem primul element.
Noul prim element este: 200
Elementele din vector: 2
C:\>
```

1208 DE CE AM UTILIZAT INSTRUCȚIUNEA USING NAMESPACE STD



În secțiunea 1207, ați scris programul *vector1.cpp*, care crea un container vector simplu și manevra obiectele sale componente. Deși întregul program este explicat pe îndelete, linia următoare de cod poate că vi s-a părut inutilă sau nelalocul ei:

```
using namespace std;
```

După cum știți, instrucțiunea *std* permite programelor dumneavoastră să acceseze nume de variabile dintr-un spațiu de nume (*namespace*) dat. În acest caz particular, instrucțiunea *using* permite prgramului dumneavoastră să acceseze variabilele și clasele din spațiul de nume *std*, care e numele de spațiu standard pentru biblioteca standard de șabloane. De fecare dată când scrieți programe ce utilizează componente din STD, trebuie să includeți instrucțiunea *using namespace std* altfel compilatorul nu va recunoaște clasa sau clasele bibliotecii standard de șabloane utilizate în programul dumneavoastră.

1209 CONTAINERELE ASOCIATIVE ALE BIBLIOTECII STANDARD DE ȘABLOANE



Un container asociativ este un container de dimensiuni variabile ce conține modalități de regăsire eficientă a elementelor (valorilor), bazându-se pe *chei*. O *cheie* este o valoare de sortare ce poate sau nu să fie valoarea reală pe care containerul o utilizează ca index al obiectelor sale. Containerele asociative acceptă inserarea și extragerea de elemente, dar diferă de containerele secvențiale prin aceea că cele asociative nu pun la dispoziție un mecanism pentru inserarea unui element la o anumită poziție. Ca și în cazul celorlalte containere, elementele dintr-un container asociativ sunt de tipul *tip_valoare*. În plus, fiecare element dintr-un container asociativ are o cheie de tipul *tip_cheie*.

În unele containere asociative cum ar fi *containerele asociative simple* (*set* și *multiset*), valorile *tip_cheie* și *tip_valoare* sunt identice – ceea ce înseamnă că elementele sunt propriile lor chei. În altele, cheia este o anumită parte a valorii. Deoarece containerele asociative păstrează elementele în funcție de cheile lor, este esențial ca cheia pe care containerul o asociază fiecărui element să fie imobilă (adică programul să nu poată schimba elementul). Prin urmare, în containerele asociative simple, elementele însele sunt imobile. În alte tipuri de containere asociative, cum ar fi containerele asociative perechi, elementele însele sunt mobile, dar programul nu poate modifica partea unui element care este cheia elementului.

În containerele asociative simple, unde elementele sunt chei, elementele sunt complet imobile. Prin urmare, tipurile de membri *iterator* și *const_iterator* sunt identice pentru containerele asociative simple. Însă alte tipuri de containere asociative au elemente mobile și pun la dispoziție iteratori prin intermediul cărora programul poate modifica elemente.

În anumite containere asociative, cum ar fi containerele asociative unice, specificațiile bibliotecii standard de șabloane garantează faptul că două elemente nu au aceeași cheie. În alte containere asociative, cum ar fi containerele asociative multiple, containerul va permite programului să memoreze mai multe elemente cu aceeași cheie în cadrul containerului. Pentru a înțelege mai bine containerele asociative, studiați următorul program, *primset.cpp*, care creează și manevrează un obiect de tipul *set*:

```
#include <iostream.h>
#include <set.h>

using namespace std;
typedef set<int> SET_INT;

void main(void)
  {
    SET_INT s1;
    SET_INT s2;
    SET_INT::iterator i;
    cout << "s1.insert(5)" << endl;
    s1.insert(5);
    cout << "s1.insert(10)" << endl;
    s1.insert(10);
    cout << "s1.insert(15)" << endl;
    s1.insert(15);
    cout << "s2.insert(2)" << endl;
    s2.insert(2);
    cout << "s2.insert(4)" << endl;
    s2.insert(4);
    cout << "swap(s1,s2)" << endl;
    swap(s1,s2);
    // Afiseaza: 2,4
    for (i=s1.begin();i!=s1.end();i++)
      cout << "s1 are pe " << *i << " in setul sau." << endl;
   // Afiseaza: 5,10,15
    for (i=s2.begin();i!=s2.end();i++)
      cout << "s2 are pe " << *i << " in setul sau." << endl;
```

```
   cout << "s1.swap(s2)" << endl;
   s1.swap(s2);
   // Afiseaza: 5,10,15
   for (i=s1.begin();i!=s1.end();i++)
     cout << "s1 are pe " << *i << " in setul sau." << endl;
  // Afiseaza: 2,4
   for (i=s2.begin();i!=s2.end();i++)
     cout << "s2 are pe " << *i << " in setul sau." << endl;
 }
```

Programul creează două *seturi*, atribuind primului *set* valorile 5,10 și 15, iar celui de-al doilea, valorle 2 și 4. Programul definește de asemenea iteratorul *i*, pe care îl utilizează după aceea pentru a traversa *seturile*. În cadrul programului *primset.cpp*, codul utilizează trei funcții membre ale clasei *set*. Funcția *swap* interschimbă cele două secvențe controlate. Funcția *begin* returnează un iterator *bidirecțional* care indică primul element al secvenței. Funcția *end* returnează un iterator *bidirecțional* care indică exact dincolo de capătul secvenței. (Veți învăța mai multe despre iteratori în secțiunea 1210). Programul inserează elemente în set, apoi se deplasează prin set inversând elemente. Atunci când compilați și executați programul *primset.cpp*, ecranul dumneavoastră va afișa următoarele:

```
s1.insert(5)
s1.insert(10)
s1.insert(15)
s2.insert(2)
s2.insert(4)
swap(s1,s2)
s1 are pe 2 in setul sau
s1 are pe 4 in setul sau
s2 are pe 5 in setul sau
s2 are pe 10 in setul sau
s2 are pe 15 in setul sau
s1.swap(s2)
s1 are pe 5 in setul sau
s1 are pe 10 in setul sau
s1 are pe 15 in setul sau
s2 are pe 2 in setul sau
s2 are pe 4 in setul sau
C:\>
```

1210 ITERATORII



Un factor cheie în proiectul bibliotecii standard de șabloane este utilizarea pe scară largă a iteratorilor în cadrul definițiilor containerelor, care generalizează pointerii limbajului C++ ca intermediari între algoritmi și containere. Biblioteca standard de șabloane definește cinci categorii de iteratori. Clasificarea este de asemenea principalul ghid în extinderea bibliotecii pentru a include noi algoritmi care să lucreze cu containerele bibliotecii standard de șabloane sau de a cuprinde noi containere la care să puteți aplica mare parte din algoritmii generici ai bibliotecii standard de șabloane.

Sunt trei lucruri pe care trebuie să le luați în seamă atunci când încercați să determinați ce algoritmi să utilizați și cu ce containere și iteratori:

- Biblioteca standard de șabloane clasifică iteratorii în cinci categorii: *de avansare, de intrare, de ieșire, bidirecționali* și *de acces aleator.*
- Fiecare descriere a unei clase container include tipurile de iteratori pe care le prevede acea clasă.
- Fiecare descriere a unui algoritm generic cuprinde categoriile de iteratori și containere cu care lucrează algoritmul generic.

Tabelul 1210.1 definește cele cinci categorii de iteratori.

Tipul de iterator	Descriere
de avansare (forward)	Prevăzuți pentru traversarea unidirecțională a unei secvențe pe care programele dumneavoastră o vor exprima cu operatorul de incrementare (++)
de intrare (input)	Similari iteratorilor *de avansare* prin aceea că programele dumneavoastră îi pot utiliza pentru a introduce date într-un container. Însă *iteratorii de intrare* pot să nu accepte toate proprietățile iteratorilor *de avansare.*
de ieșire (output)	Similari iteratorilor *de avansare* prin aceea că programele dumneavoastră îi pot utiliza pentru a extrage date dintr-un container. Însă iteratorii *de ieșire* pot să nu accepte toate proprietățile iteratorilor *de avansare.*
bidirecționali (bi-directional)	Prevăzuți pentru traversarea în ambele direcții, pe care programele dumneavoastră o vor exprima cu ++ (înainte) și – (înapoi). Rețineți că dacă iteratorul indică spre capătul containerului în direcția înainte, incrementarea iteratorului cu ++ va muta iteratorul în direcția celuilalt capăt al containerului.
cu acces aleator (random-access)	Prevăzuți pentru traversarea bidirecțională a unei secvențe. În plus, iteratorii *cu acces aleator* pun la dispoziție „salturi mari" bidirecționale într-o secvență, pe care le veți exprima ca adunare de întregi, scădere de întregi, scădere de iteratori sau comparații, după cum detaliază tabelul 1210.2.

Tabelul 1210.1 *Cele patru tipuri de iteratori ai bibliotecii standard de șabloane.*

După cum indică tabelul 1210.1, iteratorii *de acces aleator* vă permit să utilizați câteva tehnici diferite pentru a vă deplasa printr-o secvență. Tabelul 1210.2 detaliază tenicile pe care programul dumneavoastră le poate utiliza cu iteratorii *de acces aleator* pentru traversarea unei liste.

Tehnica	Explicație
adunare de întregi	Puteți efectua adunări și scăderi de întregi cu un iterator *de acces aleator* utilizând formele *r+=n* și *r-=n* (unde *r* este un iterator *de acces aleator,* iar *n* este un întreg). Operația va avea ca rezultat un iterator.
adunare și scădere	Puteți aduna și scădea un întreg dintr-un iterator utilizând formele *r+n* și *r-n* (unde *r* este un iterator *de acces aleator,* iar *n* este un întreg). Operația va avea ca rezultat un iterator.
scădere de iteratori	Puteți scădea un iterator dintr-un iterator utilizând forma *r-s* (unde *r* este un iterator *de acces aleator,* iar *s* un alt iterator *de acces aleator*). Operația va avea ca rezultat un iterator.

(continuare)

Tehnica	Explicație
comparații	Puteți efectua comparații cu iteratorii *de acces aleator* utilizând formele *r<s, r>s, r<=s* și *r>=s*. Comparația iteratorilor produce ca rezultat valori de tip *bool*.

Tabelul 1210.2 *Activitățile pe care programele dumneavoastră le pot efectua împreună cu iteratorii* ***de acces aleator***.

În plus, toate cele cinci categorii de iteratori pun la dispoizție și următoarele acțiuni:

- Testarea egalității cu operatorul == și a inegalității cu operatorul != (diferit).
- Dereferențierea, ceea ce înseamnă obținerea datelor obiectului de la poziția către care indică iteratorul, exprimată cu * (operatorul de dereferențiere al pointerilor)

Totuși, biblioteca standard de șabloane nu garantează că următoarele acțiuni se vor îndeplini după cum v-ați aștepta atunci când programele dumneavoastră manevrează iteratori:

Biblioteca standard de șabloane nu vă garantează că puteți salva un iterator *de intrare* sau *de ieșire* și să îl utilizați pentru a începe mai târziu avansarea pornind de la poziția sa curentă.

- Biblioteca standard de șabloane nu vă garantează că puteți atribui ulterior o valoare unui obiect pe care l-ați obținut anterior dintr-un container prin aplicarea operatorului de redirecționare al pointerilor (*) unui iterator *de intrare*.
- Biblioteca standard de șabloane nu vă garantează că puteți citi dintr-un obiect pe care l-ați obținut dintr-un container prin aplicarea operatorului * unui iterator *de ieșire*.
- Biblioteca standard de șabloane nu vă garantează că puteți testa doi iteratori *de ieșire* pentru egalitate sau inegalitate (ceea ce înseamnă că == și != pot să nu fie definite atunci când le aplicați iteratorilor de ieșire).

1211 *UN EXEMPLU DE ITERATOR*



După cum ați învățat în secțiunea 1210, programele dumneavoastră vor utiliza iteratori pentru a traversa sau a-și menține poziția curentă dintr-un obiect container. Tipuri diferite de containere acceptă tipuri diferite de iteratori, după cum detaliază tabelul 1211.

Tip de container	Tip de iterator
vector<T>::iterator	Iterator *cu acces aleator*
deque<T>::iterator	Iterator *cu acces aleator*
list<T>::iterator	Iterator *bidirecțional*
containere asociative	Toate containerele asociative utilizează iteratori *bidirecționali*

Tabelul 1211 *Tipurile de iteratori pe care le utilizează fiecare container.*.

După cum indică tabelul 1211, tipul *list* utilizează un iterator *bidirecțional*. Următorul program, *ut_iter.cpp*, utilizează un iterator *bidirecțional* pentru a traversa o listă de întregi:

```
#include <iostream.h>
#include <list.h>
using namespace std;
typedef list<int> LISTINT;
void main(void)
  {
    LISTINT listaUnu;
    LISTINT::iterator i;

    // Adauga cateva date
    listaUnu.push_front (2);
    listaUnu.push_front (1);
    listaUnu.push_back (3);

    // valori in lista 1 2 3
    for (i = listaUnu.begin(); i != listaUnu.end(); ++i)
      cout << *i << " ";
    cout << endl;

    // valori in lista 1 1 1 1
    for (i = listaUnu.end(); i != listaUnu.begin(); --i)
      cout << *i << " ";
    cout << endl;
  }
```

Programul *ut_iter.cpp* creează mai întâi tipul LISTINT și apoi creează instanța *listUnu* a acestui tip. Apoi, programul utilizează metodele *push_front* și *push_back* pentru a adăuga trei valori la listă (două la început și una la sfârșit). Apoi programul utilizează iteratorul *i* pentru a traversa lista înainte și înapoi. Atunci când compilați și executați programul *ut_iter.cpp*, ecranul dumneavoastră va afișa următorul rezultat (rețineți că prima valoare este zero deoarece funcția membru *end* obține locația de exact dincolo de sfârșitul vectorului):

```
1 2 3
0 3 2
C:\>
```

SĂ ÎNȚELEGEM MAI BINE TIPURILE DE ITERATORI DE INTRARE ȘI DE IEȘIRE AI BIBLIOTECII STANDARD DE ȘABLOANE

C/C++1212

După cum ați învățat, biblioteca standard de șabloane acceptă cinci tipuri de iteratori. Două tipuri de iteratori ce pot fi utilizați de către programele dumneavoastră pentru scopuri specifice sunt iteratorii *de ieșire* (pentru a prelua date dintr-un container și a le afișa) și iteratorii *de intrare* (pentru a obține date dintr-o altă locație și a o depozita în container). După cum ați învățat în secțiunea 1210, nici iteratorii de intrare, nici cei de ieșire nu dețin facilitățile complete oferite de iteratorii de *avansare* sau *bidirecționali*. Însă iteratorii *de intrare* și *de ieșire* sunt necesari programului dumneavoastră pentru a realiza funcțiile lor specifice și a clarifica în cadrul codului dumneavoastră ce funcții intenționați să efectueze iteratorii dumneavoastră. Limitările iteratorilor le intrare și de ieșire sunt semnificative. De

exemplu, programele dumneavoastră trebuie să utilizeze un iterator de ieșire numai pentru a returna informații dintr-un container dat, în loc de a-l utiliza pentru a plasa informații în acel container, ca mai jos:

```
LISTINT::output_iterator out;
for(out = listaUnu.begin(); out != listaUnu.end(); ++out)
    cout<< *i << "";
// Acest cod este posibil sa nu lucreze corect
for(out = listaUnu.begin(); out != listaUnu.end(); ++out)
    listaUnu.insert(1);
```

Deoarece iteratorul de ieșire nu se va actualiza corect pe parcursul unei operații de inserție, cea de-a doua buclă *for* va avea efecte imprevizibile. Similar, programele dumneavoastră trebuie să utilizeze iteratorii de intrare numai atunci când efectuează inserări într-un container și nu pentru a prelua date dintr-un container, ca mai jos:

```
LISTINT::input_iterator in;
for(in = listaUnu.begin(); in != listaUnu.end(); ++in)
    listaUnu.insert(1);
// Acest cod este posibil sa nu lucreze corect
for(in = listaUnu.begin(); in != listaUnu.end(); ++in)
    cout<< *i << „";
```

În plus față de obținerea de efecte imprevizibile atunci când efectuați o operație de ieșire cu un iterator de intrare sau o operație de intrare cu un iterator de ieșire, după ce incrementați un iterator de intrare sau de ieșire, nu veți putea compara, dereferenția sua incrementa în siguranță nici o copie a aceluiași iterator.

1213 ALTE TIPURI DE ITERATORI AI BIBLIOTECII STANDARD DE ȘABLOANE



După cum ați învățat în secțiunea 1210 și din nou în secțiunea 1212, există câteva limitări semnificative ale modului în care programele dumneavoastră pot utiliza iteratorii de intrare și pe cei de ieșire. Pentru a evita limitările iteratorilor de intrare și de ieșire, programele dumneavoastră pot utiliza iteratorii *de avansare, bidirecționali sau de acces aleator.* Un iterator *de avansare X* poate lua locul unui iterator de ieșire (pentru scriere) sau al unuia de intrare (pentru citire). În plus, puteți citi (utilizând V = *X) ceea ce tocmai ați scris (utilzând *X = V) prin intermediul unui iterator *de avansare.* Mai mult, puteți face copii multiple ale unui iterator de avansare, fiecare dintre ele putând fi dereferențiată sau incrementată independent de către program.

Pe lângă utilizarea unui iterator de avansare pentru a traversa și accesa containerele, programele dumneavoastră mai pot utiliza și un iterator *bidirecțional.* Iteratorii *bidirecționali* efectuează aproape aceleași procesări ca și iteratorii de avansare. Puteți însă și să decrementați un iterator bidirecțional, ca în instrucțiunile: *--X, X--* sau (*V* = *X--).

În fine, puteți utiliza un iterator *cu acces aleator* X în locul unuia bidirecțional. Iteratorii de acces aleator efectuează aproape aceleași procesări ca și iteratorii *bidirecționali* – vă permit

traversarea listei în aceeaşi manieră. În plus faţă de controlul bidirecţional al direcţiei, puteţi de asemenea să efectuaţi pe un iterator de acces aleator aceleaşi operaţii aritmetice de întregi pe care le puteţi face cu un pointer la un obiect. Fiind dat *N*, un obiect întreg, puteţi scrie *x[N]*, *x + N*, *x – N* şi *N+X* pentru a naviga printr-un container.

Reţineţi că un pointer la un obiect poate lua locul unui iterator de acces aleator sau al unui alt iterator.

Puteţi cu uşurinţă să sintetizaţi ierarhia categoriilor de iteratori prin vizualizarea ierarhiei iteratorilor disponibili dumneavoastră pentru fiecare acţiune în parte. De exemplu, pentru acces numai pentru scriere într-o secvenţă, puteţi utiliza oricare dintre următorii iteratori:

```
iterator de iesire ->
  iterator de avansare ->
    iterator bidirectional ->
      iterator de acces aleator
```

Săgeata spre dreapta îndică faptul că iteratorul din dreapta jos poate înlocui iteratorul din stânga sus faţă de săgeată. Prin urmare, orice algoritm care apelează un iterator de ieşire ar trebui să lucreze fără probleme şi cu un iterator de avansare, de exemplu. Totuşi, nu puteţi concluziona că orice algoritm ce apelează un iterator de avansare va lucra bine cu un iterator de ieşire (deoarece iteratorul de avansare este la dreapta jos faţă de iteratorul de ieşire).

Ierarhia iteratorilor este similară pentru accesul numai pentru citire la o secvenţă. Pentru accesul numai pentru citire, programele dumneavoastră pot utiliza oricare dintre următorii iteratori:

```
iterator de intrare->
  iterator de avansare ->
    iterator bidirectional ->
      iterator de acces aleator
```

Pentru acţiuni doar de citire, un iterator de intrare este cel mai slab dintre toate categoriile de iteratori, deoarece el poate traversa lista numai înainte şi este invalidat cu fiecare operaţie de ieşire. Din nou, fiecare alogritm ce apelează un iterator de intrare ar trebui să lucreze fără probleme şi cu un iterator de avansare sau orice alt iterator de sub el, în arborele său. Totuşi, nu puteţi concluziona că orice algoritm ce apelează un iterator de avansare va lucra bine cu un iterator de intrare.

Ierarhia iteratorilor este similară pentru accesul de scriere/citire într-o secvenţă. Pentru acces de scriere/citire la o secvenţă, programele dumneavoastră pot utiliza oricare dintre următorii iteratori:

```
iterator de avansare ->
  iterator bidirectional ->
    iterator de acces aleator
```

Reţineţi că un pointer la obiect poate întotdeauna servi pe post de iterator de acces aleator. Prin urmare, el poate servi şi în locul oricărei alte categorii de iterator, atâta timp cât acceptă fără probleme accesul de scriere/citire la secvenţa pe care o desemnează. Această „algebră" a iteratorilor este fundamentală pentru aproape toate celelalte prelucrări pe care programele dumneavoastră le vor efectua cu algoritmii şi containerele bibliotecii standard de şabloane. Este important să înţelegeţi capacităţile şi limitele fiecărei categorii de iteratori pentru a realiza modul în care containerele şi algoritmii bibliotecii standard de şabloane utilizează iteratorii.

1214 CONCEPTELE



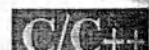
După cum aţi învăţat, este important să analizaţi clasele dumneavoastră generice pentru a determina dacă puteţi denumi în mod corect aceste clase ca fiind containere, deoarece cu o clasă container puteţi efectua anumite activităţi pe care nu le-aţi putea efectua la fel de eficient cu un obiect al unei clase simple. Totuşi este la fel de important să determinaţi alte informaţii despre clasa sau funcţia generică. O întrebare importantă la care trebuie să răspundeţi când e vorba de oricare funcţie şablon, nu numai despre algoritmii bibliotecii standard de şabloane, este ce set de tipuri poate substitui corect programul dumneavoastră în locul parametrilor formali ai şablonului. Pentru a înţelege mai bine importanţa unei substituţii corecte, parcurgeţi algoritmul *find*, definit de fişierul antet *algo.h* al bibliotecii standard de şabloane, ca mai jos:

```
InputIterator find (InputIterator first, InputIterator last,
   const T& value)
{
   while (first != last && first != value)
        ++first;
   return first;
};
```

Dacă vă uitaţi cu atenţie la definiţia algoritmului *find*, veţi vedea că el utilizează aritmetica standard de incrementare pentru lucrul cu containerele. Aceasta înseamnă că, de exemplu, puteţi substitui cu un pointer de tipul *int** sau de tipul *double** parametrul formal *InputIterator* al şablonului. Însă nu puteţi substitui *int* sau *double*, deoarece *find* utilizează expresia **first*, iar operatorul de dereferenţiere nu are sens cu un obiect de tipul *int* sau *double*. Deci, în esenţă, algoritmul *find* defineşte implicit un set de cerinţe asupra tipurilor. Prin urmare, puteţi utiliza *find* cu oricare tip ce satisface aceste cerinţe. Mai simplu, oricare tip cu care veţi substitui parametrul *InputIterator* trebuie să ofere câteva opţiuni: el trebuie să fie capabil să compare două obiecte de acel tip pentru egalitate, să incrementeze un obiect de acel tip, să dereferenţieze un obiect de acel tip pentru a obţine obiectul către care indică şi aşa mai departe.

Find nu este unicul algoritm al bibliotecii standard de şabloane care are un asemenea set de cerinţe; argumentele pentru *for_each* sau *count*, de exemplu, ca şi mulţi alţi algoritmi ai bibliotecii standard de şabloane, trebuie să îndeplinească aceleaşi cerinţe. Cei care au dezvoltat biblioteca standard de şabloane denumesc acest set important de cerinţe de tip cu numele de *concept*. Conceptul particular din cazul funcţiei *find* este conceptul de *iterator de intrare*.

1215 MODELELE



După cum aţi învăţat în secţiunea 1214, un concept defineşte un set de cerinţe pe care o funcţie şablon trebuie să le îndeplinească. În general, un tip se conformează unui concept (adică, un tip este *modelul* unui concept) dacă tipul satisface toate cerinţele conceptului. Ca urmare, puteţi spune corect că *int** este un model al conceptului de *iterator de intrare*, deoarece *int** dispune de toate operaţiile pe care cerinţele conceptului *iterator de intrare* le specifică.

Conceptele nu sunt o parte a limbajului C++. Cu alte cuvinte, nu puteți declara un concept într-un program sau să declarați că un tip particular este modelul unui concept. Totuși, conceptele și modelele lor corespunzătoare sunt componente extrem de importante ale bibliotecii standard de șabloane. Utilizarea conceptelor face posibilă scrierea programelor care separă clar interfața de implementare. De exemplu, autorul algoritmului *find* trebuie să ia în considerație numai interfața specificată de conceptul *iterator de intrare* și nu implementarea fiecărui posibil tip care se conformează acestui concept. În mod asemănător, dacă doriți să folosiți *find*, trebuie numai să vă asigurați că argumentele pe care le transmiteți sunt modele ale conceptului *iterator de intrare* (cum ar fi pointerii la întregi).

Distincția dintre interfață și implementare (care este scopul principal al tuturor funcțiilor și claselor generice, nu numai al funcțiilor și claselor bibliotecii standard de șabloane) este motivul pentru care puteți utiliza *find* și *reverse* cu elemente de tip *list*, *vector*, matrice C și multe alte tipuri de containere. Mai simplu, programarea în termeni de concepte, în loc de programarea în termeni de tipuri specifice, vă permite să reutilizați un software și să combinați componentele software.

ALGORITMII

C/C++1216

După cum ați învățat, containerul este un bloc constitutiv fundamental al bibliotecii standard de șabloane. Biblioteca standard de șabloane conține, de asemenea și o vastă colecție de algoritmi pe care programele dumneavoastră îi pot utiliza pentru a manevra datele stocate în container. De exemplu, puteți utiliza algoritmul *reverse* pentru a inversa ordinea elementelor dintr-un vector, ca mai jos:

```
reverse(v.begin(), v.end());
```

Există două puncte importante pe care trebuie să le rețineți în legătură cu acest apel al funcției *reverse*. Primul, *reverse* este o funcție globală, nu o funcție membră. Al doilea, *reverse* primește două argumente și nu unul singur – ceea ce înseamnă că *reverse* operează pe un interval de elemente dintr-un container și nu asupra containerului însuși. În acest exemplu particular, intervalul este întregul container *v* (deoarece parametrii sunt *v.begin* și *v.end*).

Motivul pentru care *reverse* este o funcție globală și acționează asupra elementelor dintr-un container și nu asupra containerului însuși este simplu. Ca și alți algoritmi ai bibliotecii standard de șabloane, *reverse* este „decuplat" de clasele container ale bibliotecii standard de șabloane. Decuplarea de clasele individuale înseamnă că puteți utiliza *reverse* pentru a inversa nu numai elementele din vectori, ci și elemente ale unor liste și chiar elemente din matrice C. Datorită decuplării algoritmului *reverse*, chiar și următorul program, *rever_tb.cpp*, este valabil:

```
#include <iostream.h>
#include <algorith.h>

using namespace std ;

void main(void)
  {
    double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
    int i;
```

```
    for (i = 0; i < 6; ++i)
      cout << "A[" << i << "] = " << A[i] << " ";
    cout << endl;
    reverse(A, A + 6);
    for (i = 0; i < 6; ++i)
      cout << "A[" << i << "] = " << A[i] << " ";
    cout << endl;
  }
```

Exemplul precedent utilizează un interval, ca și exemplul de inversare a unui vector: primul argument al funcției *reverse* este un pointer către începutul intervalului, iar al doilea argument indică cu un element dincolo de finalul intervalului. Atunci când această carte se referă la un interval de această natură (adică, un interval al cărui al doilea argument indică dincolo de finalul intervalului), intervalul va fi marcat [început, sfârșit). Intervalul utilizează notația asimetrică (adică paranteza dreaptă din stânga și paranteza din dreapta) pentru a reaminti că cele două capete sunt diferite. Primul capăt este începutul intervalului, iar cel de al doilea este cu unul peste finalul intervalului. Atunci când compilați și executați programul *rever_tb.cpp*, ecranul dumneavoastră va afișa următoarele:

```
A[0] = 1.2 A[1] = 1.3 A[2] = 1.4 A[3] = 1.5 A[4] = 1.6 A[5] = 1.7
A[0] = 1.7 A[1] = 1.6 A[2] = 1.5 A[3] = 1.4 A[4] = 1.3 A[5] = 1.2
C:\>
```

1217 UTILIZAREA ALTUI EXEMPLU DE ALGORITM AL BIBLIOTECII STANDARD DE ȘABLOANE



În secțiunea 1216 ați învățat despre algoritmii bibliotecii standard de șabloane și modul în care programele dumneavoastră pot să îi utilizeze împreună cu containerele bibliotecii standard de șabloane și cu alte tipuri de containere. În secțiunea 1214, ați învățat despre algoritmul *find* al bibliotecii standard de șabloane. Programele dumneavoastră pot utiliza algoritmul *find* al bibliotecii standard de șabloane pentru a localiza instanțe ale unui obiect conținut într-un container. Veți utiliza, în general, algoritmul *find* pentru a localiza, de exemplu, un nod individual dintr-un container de tipul clasei *list*. În plus, puteți utiliza algoritmul *find* pentru localizarea unor elemente individuale dintr-o matrice, așa cum arată următorul program, *tab_find.cpp*:

```
#include <iostream.h>
#include <algorith.h>
using namespace std;
void main(void)
  {
    const int DIM_TABLOU = 8;
    int TabInt[DIM_TABLOU] = { 1, 2, 3, 4, 4, 5, 6, 7 };
    int *locatie ; //stocheaza pozitia primului element potrivit.
    int i;
    int val = 4;
    // afiseaza continutul lui TabInt
```

```
   cout << "TabInt { ";
   for (i = 0; i < DIM_TABLOU; i++)
     cout << TabInt[i] << ", ";
   cout << "\b }" << endl;
   //Gaseste primul element din intervalul (primul, ultimul + 1)
   //care se potriveste cu val.
   locatie = find(TabInt, TabInt + DIM_TABLOU, val);
   // afiseaza elementul daca vreunul a fost intalnit
   if (locatie != TabInt + DIM_TABLOU)
   // a gasit un element potrivit
     cout << "Primul element care se potriveste cu " << val
       << " se afla la locatia " << (locatie - TabInt) << endl;
   else    // nu a fost gasit nici un element potrivit
     cout << "Secventa nu contine nici un element cu valoarea "
       <<      val << endl;
 }
```

Programul *tab_find.cpp* creează mai întâi o matrice de valori și apoi caută un anumit element din matrice. Programul utilizează algoritmul *find* pentru a determina dacă elementul se află în matrice. Atunci când compilați și executați programul *tab_find.cpp*, ecranul dumneavoastră va afișa următoarele:

```
TabInt { 1, 2, 3, 4, 5, 6, 7 }
Primul element care se potriveste cu 4 se afla la locatia 3
C:\>
```

DESCRIEREA ALGORITMILOR CUPRINȘI ÎN BIBLIOTECA STANDARD DE ȘABLOANE

C/C++ 1218

După cum ați învățat, biblioteca standard de șabloane dispune de un mare număr de algoritmi pe care programele dumneavoastră îi pot utiliza atunci când operează atât cu obiecte din biblioteca standard de șabloane, cât și cu obiecte care nu sunt în biblioteca standard de șabloane. Biblioteca standard de șabloane împarte algoritmii în patru tipuri de bază. Primul tip este setul de algoritmi *ne-modificanți*. Algoritmii ne-modificanți nu modifică containerele asupra cărora operează și caută să producă rezultate lineare. Tabelul 1218.1 listează algoritmii ne-modificanți ai bibliotecii standard de șabloane.

Algoritmi ne-modificanți

for_each	*find*	*find_if*	*adjacent_find*
find_first_of	*count*	*count_if*	*mismatch*
equal	*search*	*search_n*	*find_end*

Tabelul 1218.1 *Algoritmii ne-modificanți ai bibliotecii standard de șabloane.*

Al doilea tip de algoritmi ai bibliotecii standard de șabloane este setul de algoritmi *modificanți*. De regulă, algoritmii modificanți modifică fie natura obiectelor din cadrul containerului, fie copiază acele obiecte în alte containere. Tabelul 1218.2 listează algoritmii modificanți ai bibliotecii standard de șabloane.

Algoritmi modificanți			
copy	*copy_n*	*copy_backward*	*swap*
iter_swap	*swap_ranges*	*transform*	*replace*
replace_if	*replace_copy*	*replace_copy_if*	*fill*
fill_n	*generate*	*generate_n*	*remove*
remove_if	*remove_copy*	*remove_copy_if*	*unique*
unique_copy	*reverse*	*reverse_copy*	*rotate*
rotate_copy	*random_shuffle*	*random_sample*	*random_sample_n*
partition	*stable_partition*		

Tabelul 1218.2 *Algoritmii modificanți ai bibliotecii standard de șabloane.*

Al treilea tip important de algoritmi ai bibliotecii standard de șabloane este setul de algoritmi de *sortare.* Deși algoritmii de sortare sunt, din punct de vedere tehnic, un subset al algoritmilor modificanți, setul de algoritmi de sortare este sufuicient de extins pentru a fi tratat separat. Tabelul 1218.3 listează algoritmii de sortare ai bibliotecii standard de șabloane.

Algoritmi de sortare			
sort	*stable_sort*	*partial_sort*	*partial_sort_copy*
is_sorted	*nth_element*	*lower_bound*	*upper_bound*
equal_range	*binary_search*	*merge*	*inplace_merge*
includes	*set_union*	*set_intersection*	*set_difference*
set_symmetric_difference	*push_heap*	*pop_heap*	*make_heap*
sort_heap	*is_heap*	*min*	*max*
min_element	*max_element*	*lexicographical_compare*	
lexicographical_compare_3way		*next_permutation*	*prev_permutation*

Tabelul 1218.3 *Algoritmii de sortare ai bibliotecii standard de șabloane.*

În sfârșit, biblioteca standard de șabloane acceptă, de asemenea, un set de algoritmi *numerici generalizați.* Programele dumneavoastră pot utiliza algoritmii numerici generalizați pentru a efectua operațiuni matematice asupra unui tip necunoscut sau a unor tipuri de numere (de exemplu, un *float* și un *double* sau un *float* și un *int*), în funcție de acel algoritm și de scopul său. Tabelul 1228.4 listează algoritmii numerici generalizați ai bibliotecii standard de șabloane.

Algoritmi numerici generalizați		
iota	*accumulate*	*inner_product*
partial_sum	*adjacent_difference*	*power*

Tabelul 1218.4 *Algoritmii numerici generalizați ai bibliotecii standard de șabloane.*

Desigur, lista algoritmilor pe care biblioteca standard de șabloane îi definește este cuprinzătoare. Din păcate, această carte nu va aborda toți algoritmii. Însă, puteți parcurge în documentația compilatorului dumneavoastră sau documentația bibliotecii standard de șabloane pentru mai multe informații despre orice algoritm listat în această secțiune.

STUDIEREA ALGORITMULUI FOR_EACH

C/C++ 1219

După cum aţi învăţat, biblioteca standard de şabloane defineşte diferite tipuri de algoritmi. Deşi aţi utilizat deja algoritmul ne-modificant *find*, este util pentru dumneavoastră să examinaţi un al doilea exemplu de program, care utilizează algoritmul ne-modificant *for_each*. Algoritmul *for_each* apelează o funcţie *Func1* pentru fiecare element din intervalul [primul, ultimul) şi nu returnează nici o valoare, ca mai jos:

```
void for_each(primul, ultimul, func1);
```

Algoritmul *for_each* nu modifică nici un element din secvenţă. De exemplu, următorul program, *putere_3.cpp*, utilizează algoritmul *for_each* pentru a accesa fiecare element dintr-un vector şi a afişa cubul acelui element:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>
using namespace std;

void AfisCub(int n)
  {
    // afiseaza cubul intregului n
    cout << "Cubul lui " << n << " este " << n * n * n << endl;
  }

void main(void)
  {
    const int DIM_VECTOR = 8;
    typedef vector<int> VectorInt; // Defineste un vector de
                                    // intregi
    typedef VectorInt::iterator ItVectorInt;
       // Defineste un tip de iterator

    VectorInt Numere(DIM_VECTOR);  // vector continand numere
    ItVectorInt start, final, it;  // iteratori
    int i;

    for (i = 0; i < DIM_VECTOR; i++) // Initializeaza vectorul
                                      // Numere
      Numbers[i] = i + 1;

    start = Numere.begin(); // locatia primului element din
                            // Numere
    final = Numere.end() ;
       // un element dupa locatia ultimului element din Numere

    cout << "Numere { "; // afiseaza continutul lui Numere
    for(it = start; it != final; it++)
      cout << *it << " " ;
    cout << " }\n" <<endl ;

     //pentru fiecare element din intervalul (primul, ultimul),
```

```
   // afiseaza cubul elementului
   for_each(start, final, AfisCub);
}
```

Programul *putere_3.cpp* creează inițial vectorul *VectorInt*, apoi atribuie o valoare fiecărui obiect din vector. După afișarea valorilor vectorului, programul utilizează algoritmul *for_each* pentru a afișa cubul fiecărui element al vectorului. Atunci când compilați și executați programul *putere_3.cpp*, ecranul dumneavoastră va afișa următorul rezultat:

```
Numere { 1 2 3 4 5 6 7 8 }
Cubul lui 1 este 1
Cubul lui 2 este 8
Cubul lui 3 este 27
Cubul lui 4 este 64
Cubul lui 5 este 125
Cubul lui 6 este 216
Cubul lui 7 este 343
Cubul lui 8 este 512
C:\>
```

1220 STUDIEREA ALGORITMULUI GENERATE_N C/C++

După cum ați învățat în secțiunea 1218, biblioteca standard de șabloane conține un mare număr de algoritmi pe care programele dumneavoastră îi pot utiliza pentru a opera cu containere și cu alte obiecte. Unul dintre cei mai deosebiți algoritmi modificanți pe care îi acceptă biblioteca standard de șabloane este algoritmul *generate_n*, care completează fiecare obiect dintr-un interval de obiecte din interiorul unui container cu valoarea returnată a unei funcții *generator*. Funcția generator returnează o valoare pe care algoritmul o plasează în obiecte. Pentru a înțelege mai bine modul în care funcția generator returnează valoarea, studiați următorul program, *gen_fib.cpp*, care utilizează algoritmul *generate_n* pentru a plasa numere din secvența Fibonacci într-un vector:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>

using namespace std;

  //returneaza urmatorul numar Fibonacci din seria Fibonacci.
int Fibonacci(void)
  {
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2;
    f1 = f2;
    f2 = r;
    return f1;
  }

void main(void)
```

```
{
  const int DIM_VECTOR = 15;
  //Defineste o clasa sablon vector de intregi
  typedef vector<int> VectorInt;
  //Defineste un iterator pentru clasa sablon vector
  typedef VectorInt::iterator ItVectorInt;
  VectorInt Numere(DIM_VECTOR); //vector continand numere
  ItVectorInt start, final, it;
  int i;
  //Initializeaza vectorul Numere
  for(i = 0; i < DIM_VECTOR; i++)
    Numere[i] = i * i;
  start = Numere.begin(); // locatia primului element din
                          // Numere
  end = Numbers.end();
     // un element dupa locatia ultimului element din Numere
  cout << "Inaintea apelarii lui generate_n" << endl;
  //afiseaza continutul lui Numere
  cout << "Numere { ";
  for (it = start; it != final; it++)
    cout << *it << " ";
  cout << " }\n" << endl;
  //completeaza intervalul specificat cu o serie de
  //numere Fibonacci utilizand functia Fibonacci
  generate_n(start + 5, Numere.size() - 5, Fibonacci);
  cout << "Dupa apelarea lui generate_n" << endl;
  //afiseaza continutul lui Numere
  cout << "Numere {";
  for (it = start; it != final; it++)
    cout << *it << " ";
  cout << " }\n" << endl;
}
```

Programul *gen_fib.cpp* inițializează vectorul *numere* pentru a păstra 14 valori, fiecare dintre ele conținând pătratul indicelui valorii din cadrul vectorului. De exemplu, valoarea de la indicele 1 conține 1 (1*1) și valoarea de la indicele 12 conține 144 (12*12). Apoi, programul înlocuiește numerele din cadrul vectorului începând de la al șaselea număr, pe care îl înlocuiește cu primul număr din secvența Fibonacci. Programul înlocuiește fiecare număr rămas din vector cu următorul număr din secvența Fibonacci. Atunci când compilați și executați programul *gen_fib.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Inaintea apelarii lui generate_n
Numere { 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 }

Dupa apelarea lui generate_n
```

```
Numere { 0 1 4 9 16 1 1 2 3 5 8 13 21 34 55 }
C:\>
```

1221 ALGORITMUL RANDOM_SHUFFLE

C/C++

Pe măsură ce programele dumneavoastră devin mai complexe, veți avea uneori nevoie să rearanjați informațiile dintr-un container într-o ordine oarecare. În general, veți efectua astfel de procesări atunci când scrieți jocuri sau alte programe de claculator care simulează răspunsuri inteligente ale calculatorului. Pentru a simplifica astfel de procesări, biblioteca standard de șabloane pune la dispoziție algoritmul *random_shuffle*. Algoritmul *random_shuffle* aranjează elementele unei secvențe (de la primul până la ultimul) într-o ordine aleatoare. Algoritmul *random_shuffle* utilizează fie un generator intern de numere aleatoare pentru a genera indicii elementelor pe care le interschimbă, fie alimentează generatorul de numere aleatoare cu numărul de elemente cuprinse în container, în funcție de modul de apelare. Ambele versiuni ale algoritmului *random_shuffle* utilizează operatorul = pentru a efectua interschimbări. Pentru a înțelege mai bine prelucrarea efectuată de algoritmul *random_shuffle*, analizați programul *shuffle.cpp*, arătat mai jos:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>

using namespace std;

void main(void)
  {
    const int DIM_VECTOR = 8 ;
    // Defineste un vector de int
    typedef vector<int> VectorInt ;
    //Defineste un iterator pentru vector
    typedef VectorInt::iterator ItVectorInt ;
    VectorInt Numere(DIM_VECTOR) ;

    ItVectorInt start, final, it ;

    // Initializeaza vectorul Numere
    Numere[0] = 4 ;
    Numere[1] = 10;
    Numere[2] = 70 ;
    Numere[3] = 30 ;
    Numere[4] = 10;
    Numere[5] = 69 ;
    Numere[6] = 96 ;
    Numere[7] = 100;

    start = Numere.begin() ; // locatia primului element din
                             // Numere
    final = Numere.end() ;   // cu unul dincolo de locatia
       // ultimului element din Numere
    cout << "Inaintea apelarii lui random_shuffle\n" <<endl ;
    // afiseaza continutul lui Numere
```

```
    cout << "Numere { " ;
    for(it = start; it != final; it++)
      cout << *it << " " ;
    cout << " }\n" <<endl ;
    // aranjeaza elementele in ordine aleatoare
    random_shuffle(start, final) ;
    cout << "Dupa apelarea lui random_shuffle\n" <<endl ;

    cout << "Numere { " ;
    for(it = start; it != final; it++)
      cout << *it << " " ;
    cout << "\b }\n" <<endl ;
}
```

Mai întâi, programul *shuffle.cpp* creează un vector şi-l completează cu o serie de numere. Apoi, programul apelează algoritmul *random_shuffle* şi îi indică să amestece toate numerele din vector. În final, programul afişează din nou numerele, după ce a încheiat algoritmul *random_shuffle*. Atunci când compilaţi şi executaţi programul *shuffle.cpp*, ecranul va afişa un rezultat asemănător cu următorul:

```
Inaintea apelarii lui random_shuffle
Numere { 4 10 70 30 10 69 96 100 }

Dupa apelarea lui random_shuffle
Numere { 96 4 69 70 10 10 30 100 }
C:\>
```

UTILIZAREA ALGORITMULUI PARTIAL_SORT_COPY

C/C++1222

După cum aţi învăţat în secţiunea 1218, biblioteca standard de şabloane acceptă mulţi algoritmi cu diferite funcţii. În secţiunile precedente aţi scris scurte programe care utilizau atât algoritmi modificanţi, cât şi nemodificanţi ai bibliotecii standard de şabloane. În secţiunea 1223 veţi utiliza algoritmul *merge* pentru a combina valorile din doi vectori. După cum aţi învăţat, funcţia *merge* este unul dintre algoritmii de sortare ai bibliotecii standard de şabloane. În această secţiune veţi utiliza alt algoritm de sortare, numit *partial_sort_copy*.

Algoritmul *partial_sort_copy* sortează cele mai mici *N* elemente, unde *N = min((ultimul1 – primul1), (ultimul2 – primul2))*, ale secvenţei [primul1, ultimul1) şi copiază rezultatul în secvenţa *[primul2, primul2 + N]*. Cu alte cuvinte, algoritmul *partial_sort_copy* sortează containerul într-un container temporar, în ordine ascendentă. Algoritmul *partial_sort_copy* determină apoi cel mai mic număr din containerul temporar (primul număr). Apoi copiază acel număr de obiecte din containerul sortat temporar într-un al treilea container, permanent. De exemplu, dacă o matrice conţine valorile 15, 25, 7, 8, 10, 12 şi 2, matricea va apărea iniţial ca în figura 1222.1.

15	25	7	8	10	12	2

***Figura 1222.1** Matricea înainte de sortare sau copiere.*

Algoritmul *partial_sort_copy* va sorta apoi matricea și o va stoca într-o matrice temporară, ca în figura 1222.2.

2	7	8	10	12	15	25

Figura 1222.2 *Matricea temporară sortată.*

Apoi funcția *partial_sort_copy* determină cel mai mic număr din rezultatul sortat (în acest caz, 2) și copiază primele două valori din matricea sortată temporară într-o copie permanentă, rezultând două matrice, ca în figura 1222.3.

15	25	7	8	10	12	2

2	7

Figura 1222.3 *Cele două matrice rezultate din apelul funcției* ***partial_sort_copy.***

Pentru a înțelege mai bine acțiunile pe care algoritmul *partial_sort_copy* le efectuează, studiați următorul program, *ps_copy.cpp*, care efectuează pașii descriși în ilustrațiile precedente:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>

using namespace std;

void main(void)
  {
    const int DIM_VECTOR = 8;

    //Defineste o clasa sablon vector de int
    typedef vector<int> VectorInt;
    //Defineste un iterator pentru clasa sablon vector de int
    typedef VectorInt::iterator ItVectorInt;
    VectorInt Numere(DIM_VECTOR);
    VectorInt Rezultat(4);
    ItVectorInt start, final, it;
    //Initializeaza vectorul Numere
    Numere[0] = 4;
    Numere[1] = 10;
    Numere[2] = 70;
    Numere[3] = 30;
    Numere[4] = 10;
    Numere[5] = 69;
    Numere[6] = 96;
```

```
   Numere[7] = 7;
   start = Numere.begin(); // locatia primului element din
                           // Numere
   final = Numere.end();   // cu unul dincolo de locatia
                           // ultimului element din Numere
   cout << "Inainte de apelarea lui partial_sort_copy\n" << endl;
   //afiseaza continutul lui Numere
   cout << "Numere { ";
   for (it = start; it != final; it++)
     cout << *it << " ";
   cout << " }\n" << endl;
   //sorteaza cele mai mici 4 elemente din Numere
   //si copiaza rezultatele in Rezultat
   partial_sort_copy(start, final, Rezultat.begin(),
     Rezultat.end());
   cout << "Dupa apelarea lui partial_sort_copy\n" << endl;
   cout << "Numere { ";
   for(it = start; it != final; it++)
     cout << *it << " ";
   cout << " }\n" << endl;
   cout << "Rezultat { ";
   for(it = Rezultat.begin(); it != Rezultat.end(); it++)
     cout << *it << " ";
   cout << "\b }\n" << endl;
 }
```

Atunci când compilaţi şi executaţi programul *ps_copy.cpp*, el va crea un vector de 8 elemente întregi. Apoi, el va atribui valori acestor elemente. Pentru a arăta activitatea sa, *ps_copy.cpp* afişează elementele vectorului înainte de apelarea algoritmului *partial_sort_copy*. În cadrul funcţiei *partial_sort_copy*, algortimul determină că 4 este cel mai mic număr şi copiază primele patru valori din matricea sortată în matricea de ieşire. Atunci când compilaţi şi executaţi programul *ps_copy.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Inainte de apelarea lui partial_sort_copy
Numere { 4 10 70 30 10 69 96 7 }

Dupa apelarea lui partial_sort_copy
Numere { 4 10 70 30 10 69 96 7 }
Rezultat { 4 7 10 10 }
C:\>
```

ALGORITMUL MERGE

C/C++1223

În secţiunea 1222 aţi utilizat algoritmul *partial_sort_copy* pentru a crea o copie parţial sortată a unui vector. În această secţiune, veţi utiliza algoritmul *merge* pentru a fuziona doi vectori sortaţi într-un al treilea vector sortat ce va conţine toate valorile din cei doi vectori originali. Veţi utiliza algoritmul *merge* în programele dumneavoastră după cum arătăm în următorul prototip:

```
merge(primul1, ultimul1, primul2, ultimul2, rezultat);
```

Algoritmul *merge* combină două secvențe sortate: (*primul1, ultimul1*) și (*primul2, ultimul2*) într-o singură secvență sortată începând de la iteratorul *rezultat*. Algoritmul *merge* presupune că intervalele (*primul1 .. ultimul1*) și (*primul2 .. ultimul2*) au fost sortate în prealabil cu ajutorul operatorului „mai mic decât". Dacă ambele intervale conțin valori egale, *merge* va stoca mai întâi valorile din primul interval în intervalul rezultat. Pentru a înțelege mai bine activitatea algoritmului *merge*, studiați următorul program, *merge_2v.cpp*:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>
#include <list.h>
#include <deque.h>
using namespace std;
void main(void)
{
   const int MAX_ELEMENTE = 8;
   typedef vector<int> VectorInt;
   //Defineste o clasa sablon de vectori de int
   typedef VectorInt::iterator ItVectorInt;
   //Defineste un tip iterator
   typedef list<int> ListaInt;
   //Defineste o clasa sablon lista de intregi
   typedef ListaInt::iterator ListaIntIt;
   //Defineste un tip iterator
   typedef deque<int> IntCoada;
   //Defineste o clasa sablon coada(deque) de intregi
   typedef IntCoada::iterator IntCoadaIt;
   //Defineste un tip iterator
   VectorInt VectorNumere(MAX_ELEMENTE);
   ItVectorInt startv, finalv, itv;
   ListaInt ListaNumere;
   ListaIntIt primul, ultimul, itl;
   IntCoada CoadaNumere(2 * MAX_ELEMENTE);
   IntCoadaIt itd;
   //Initializeaza vectorul VectorNumere
   VectorNumere[0] = 4;
   VectorNumere[1] = 10;
   VectorNumere[2] = 70;
   VectorNumere[3] = 10;
   VectorNumere[4] = 30;
   VectorNumere[5] = 69;
   VectorNumere[6] = 96;
   VectorNumere[7] = 100;
   startv = VectorNumere.begin();
     //locatia primului element din VectorNumere
```

```
finalv = VectorNumere.end();
     //cu unu dupa ultimul element din VectorNumere
     //sorteaza VectorNumere, merge impune ca secventele sa
     // fie sortate
sort(startv, finalv);
//afiseaza continutul lui VectorNumere
cout << "VectorNumere { ";
for(itv = startv; itv != finalv; itv++)
  cout << *itv << " ";
cout << "\b }\n" << endl;
//Initializeaza lista ListaNumere
for(int i = 0; i < MAX_ELEMENTE; i++)
  ListaNumere.push_back(i);
primul = ListaNumere.begin();
   //locatia primului element din ListaNumere
ultimul = ListaNumere.end();
  //cu unu dupa ultimul element din ListaNumere
//afiseaza continutul ListaNumere
cout << "ListaNumere { ";
for(itl = primul; itl != ultimul; itl++)
  cout << *itl << " ";
cout << " }\n" << endl;

//combina elementele din VectorNumere si din ListaNumere si
//plaseaza rezltatele in CoadaNumere
merge(startv, finalv, primul, ultimul, CoadaNumere.begin());
//afiseaza continutul din CoadaNumere
cout << "Dupa apelarea lui merge\n" << endl;
cout << "CoadaNumere { " ;
for (itd = CoadaNumere.begin(); itd != CoadaNumere.end(); itd++)
  cout << *itd << " ";
cout << "\b }\n" << endl;
}
```

Programul *merge_2v.cpp* mai întâi creează şi iniţializează vectorul *VectorNumere*. Apoi el apelează funcţia *sort* pentru a plasa numerele vectorului în ordine ascendentă. Apoi programul creează lista *ListaNumere* pe care codul o creează ca fiind în întregime ordonată. După ce *merge_2v.cpp* creează lista *ListaNumere*, el apelează funcţia *merge* şi palsează rezultatul în containerul *CoadaNumere*. Pentru a arăta că şi-a încheiat activitatea, *merge_2v.cpp* afişează numerele cuprinse în containerul *CoadaNumere*. Atunci când compilaţi şi executaţi programul *merge_2v.cpp*, ecranul dumneavoastră va afişa următorul rezultat:

```
VectorNumere { 4 10 10 30 69 70 96 100 }
ListaNumere { 0 1 2 3 4 5 6 7 }

Dupa apelarea lui merge
CoadaNumere { 0 1 2 3 4 4 5 6 7 10 10 30 69 70 96 100 }
C:\>
```

1224 ALGORITMUL INNER_PRODUCT



În secţiunea 1223 aţi utilizat algoritmul *merge* pentru a fuziona un vector sortat şi o listă sortată într-o coadă (*deque*) sortată care conţinea toate valorile din ambele obiecte iniţiale. De fapt, în secţiunile recente, aţi utilizat algoritmi ai bibliotecii standard de şabloane de trei din cele patru mari tipuri: modificanţi, ne-modificanţi şi de sortare. În această secţiune veţi utiliza unul dintre algoritmii numerici generalizaţi, algoritmul *inner_product*. Algoritmul *inner_product* înmulţeşte *N* elemente din două containere unul cu celălalt şi returnează suma rezultatelor înmulţirilor (aceasta se numeşte produs intern). De exemplu, când calculaţi produsul intern a două matrice începând cu al doilea element şi continuând cu trei elemente, valoarea *inner_product* va fi conform cu ceea ce arată figura 1224.1.

1	2	3	4	5
5	4	3	2	1

= (2*4)+(3*3)+(2*4)
= 8+9+8
= 25

Figura 1224.1 *Cum calculează algoritmul* ***inner_product*** *„suma produselor" – rezultatul produsului intern pentru două matrice.*

Algoritmul *inner_product* mai acceptă şi o a doua implementare, supraîncărcată, care este produsul sumelor valorilor interne. Utilizând exemplul arătat în Figura 1224.1, veţi determina produsul sume produsului intern, ca în figura 1224.2.

1	2	3	4	5
5	4	3	2	1

= (2+4)(3+3)(4+2)
= (6)(6)(6)
= 216.

Figura 1224.2 *Cum calculează algoritmul* ***inner_product*** *„produsul sumelor" – rezultatul produsului intern pentru două matrice.*

Pentru a înţelege mai bine activitatea pe care o efectuează algoritmul *inner_product*, studiaţi următorul program, *in_prod.cpp*:

```
#include <iostream.h>
#include <vector.h>
#include <numeric.h>
#include <iterator.h>
```

```
using namespace std;
typedef vector<float> MatriceFloat;
typedef ostream_iterator<float, char, char_traits<char>>
FloatOstreamIt;
void main(void)
  {
    FloatOstreamIt itOstream(cout, " ");
    //Initializeaza matricele
    TablouFloat rgF1, rgF2;
    for (int i=1; i<=5; i++)
      {
        rgF1.push_back(i);
        rgF2.push_back(i*i);
      };
    //afiseaza Matricele
    cout << "Matrice 1: ";
    copy(rgF1.begin(), rgF1.end(), itOstream);
    cout << endl;
    cout << "Matrice 2: ";
    copy(rgF2.begin(), rgF2.end(), itOstream);
    cout << endl;
    // Aceasta este suma produselor (S.P.) a elementelor
    // corespondente
    float ip1 = inner_product(rgF1.begin(), rgF1.end(),
      rgF2.begin(), 0);
    cout << "Produsul intern (S.P.) al matricelor 1 si 2 este "
      << ip1 << endl;
    // Acesta este produsul sumelor (P.S.) al elementelor
    // corespondente
    float ip2 = inner_product(rgF1.begin(), rgF1.end(),
      rgF2.begin(), 1, multiplies<float>(), plus<float>());
    cout << "Produsul intern (P.S.) al matricelor 1 si 2 este "
      << ip2 << endl;
  }
```

Programul *in_prod.cpp* începe cu iniţializarea a doi vectori de tipul *float* şi completarea vectorilor cu valori - primul cu o secvenţă liniară şi al doilea cu pătratele primei secvenţe. Programul afişează cei doi vectori pentru analiza dumneavoastră. Apoi, programul calculează produsul intern ca sumă a produselor (S.P.) şi continuă cu calcularea produsului intern ca produs al sumelor (P.S.). Atunci când compilaţi şi executaţi programul *in_prod.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Matrice 1: 1 2 3 4 5
Matrice 2: 1 4 9 16 25
Produsul intern (S.P.) al matricelor 1 si 2 este 225
Produsul intern (P.S.) al matricelor 1 si 2 este 86400
C:\>
```

1225 VECTORII



După cum aţi învăţat, un *vector* este un container secvenţial care acceptă acces aleator la elemente, inserări şi eliminări de elemente în timp constant la capătul containerului (adică, cantitatea de timp pe care programul o utilizează pentru a insera elemente nu variază în funcţie de dimensiunea vectorului) şi inserări şi eliminări de elemente de la începutul sau mijlocul containerului cu durată lineară (cantitatea de timp neceesară programului pentru a insera elemente variază în funcţie de dimensiunea vectorului). Numărul de elemente dintr-un vector poate varia în mod dinamic. După cum veţi învăţa, din cauză că vectorul poate realoca mai multă memorie pentru sine însuşi, modul său de gestionare a memoriei este automatic. Clasa *vector* este cea mai simplă dintre clasele container ale bibliotecii standard de şabloane şi, în multe cazuri, cea mai eficientă. După cum aţi învăţat, un *vector* este similar unei matrice. Următorul fragment de cod, de exemplu, arată cum se declară şi se utilizează un vector:

```
vector<int> V;
V.insert(V.begin(), 3);
```

Prima linie declară un vector *V* de întregi, care nu conţine nici un element. Cea de-a doua linie inserează valoarea 3 la începutul vectorului. În plus faţă de funcţiile membre *insert* şi *begin*, clasa *vector* acceptă mai multe funcţii membre. Tabelul 1225 prezintă funcţiile membre ale clasei *vector* şi o scurtă descriere a fiecăreia.

Membru	Descriere
value_type	Tipul obiectului, *T*, stocat în vector.
pointer	Pointer la *T*.
reference	Referinţă la *T*.
const_reference	Referinţă *const* la *T*.
size_type	Un tip integral fără semn.
difference_type	Un tip integral cu semn.
iterator	Definiţia tipului iteratorului de bază pe care programul dumneavoastră trebuie să-l utilizeze pentru a itera un vector.
const_iterator	Definiţia tipului *Const* al iteratorului de bază pe care programul dumneavoastră trebuie să-l utilizeze pentru a itera un vector.
reverse_iterator	Definiţia tipului iteratorului de bază pe care programul dumneavoastră trebuie să-l utilizeze pentru a itera în sens invers un vector.
const_reverse_iterator	Definiţia tipului *Const* al iteratorului de bază pe care programul dumneavoastră trebuie să-l utilizeze pentru a itera în sens invers un vector.

Membru	**Descriere**
iterator begin()	Returnează un iterator indicând către începutul vectorului.
iterator end()	Returnează un iterator indicând către finalul vectorului.
const_iterator begin() const	Returnează un *const_iterator* indicând către începutul vectorului.
const_iterator end() const	Returnează un *const_iterator* indicând către finalul vectorului.
reverse_iterator rbegin()	Returnează un *reverse_iterator* indicând către începutul vectorului inversat.
reverse_iterator rend()	Returnează un *reverse_iterator* indicând către finalul vectorului inversat.
const_reverse_iterator rbegin() const	Returnează un *const_reverse_iterator* indicând către începutul vectorului inversat.
const_reverse_iterator rend() const	Returnează un *const_reverse_iterator* indicând către finalul vectorului inversat.
size_type size() const	Returnează dimensiunea în elemente a vectorului.
size_type max_size() const	Returnează cea mai mare dimensiune posibilă pentru vector în elemente.
size_type capacity() const	Numărul de elemente pentru care vectorul a alocat memorie. Valoarea pe care o returnează *capacity* este întotdeauna mai mare sau egală cu *size*.
bool empty() const	*True* dacă dimensiunea vectorului este 0.
reference operator[ ](size_type n)	Returnează al *n*-lea element al containerului.
const_reference operator[ ](size_type n) const	Returnează o reprezentare valoare constantă a celui de-al *n*-lea element al containerului.
vector()	Creează un vector gol.
vector(size_type n)	Creează un vector cu *n* elemente.
vector(size_type n, const T& t)	Creează un vector cu *n* copii ale obiectului *t*.
vector(const vector&)	Constructorul copie.
vector(InputIterator, InputIterator)	Creează un vector cu o copie a unui interval.
~vector()	Funcția destructor.
vector& operator=(const vector&)	Operatorul de atribuire.

(continuare)

Membru	Descriere
void reserve(size_t n)	Dacă *n* este mai mic sau egal cu *capacity*, acest apel nu are efect. Altfel, este o cerere pentru alocarea de memorie suplimentară. Dacă cererea este satisfăcută, funcția stabilește apoi *capacity* ca fiind mai mare sau egală cu *n*; altfel, *capacity* rămâne neschimbată. În acest caz, proprietatea *size* a vectorului rămâne neschimbată.
reference front()	Returnează primul element din vector.
const_reference front() const	Returnează o reprezentare de valoare constantă a primului element din vector.
reference back()	Returnează ultimul element din vector.
const_reference back() const	Returnează o reprezentare de valoare constantă a ultimului element din vector.
void push_back(const T&)	Inserează un nou element la finalul vectorului.
void pop_back()	Elimină ultimul element din vector.
void swap(vector&)	Interschimbă conținutul a doi vectori, presupunând că vectorii sunt de tipuri compatibile. Cu alte cuvinte, dacă doi vectori Z și Y au amândoi valori întregi, puteți interschimba valorile celor doi vectori cu instrucțiunea *swap*.
iterator insert(iterator pos, const T& x)	Inserează *x* după poziția *pos* din vector.
void iterator insert(iterator pos, InputIterator f, InputIterator l)	Inserează intervalul [f,l) înaintea poziției *pos* din vector.
void insert(iterator pos, size_type n, const T& x)	Inserează *n* copii ale lui *x* înaintea poziției *pos* din vector.
iterator erase(iterator pos)	Elimină elementul de la poziția *pos* din vector.
iterator erase(iterator first, iterator last)	Elimină intervalul [f,l) din vector.
bool operator==(const vector&, const vector&)	Testează egalitatea a doi vectori. Aceasta este o funcție globală, nu o funcție membră.
bool operator<(const vector&, const vector&)	Comparație lexicografică. Aceasta este o funcție globală, nu o funcție membră.

Tabelul 1225 *Membrii clasei* ***vector.***

După cum ați învățat în secțiunile precedente, biblioteca standard de șabloane conține atât de mulți algoritmi încât pentru a fi descriși în totalitate ar fi nevoie de tot spațiul acestei cărți. La fel, clasa *vector* are atât de mulți membri încât nu pot fi descriși toți. Secțiunile care urmează se vor referi la unii dintre cei mai des utilizați membri ai clasei *vector*, ca și la alte clase ale bibliotecii standard de șabloane. Atunci când veți lucra cu alte clase în secțiunile ce

urmează, veți observa că aceste clase, la rândul lor, au atâția membri încât nu pot fi cuprinși cu toții în această carte.

UTILIZAREA UNUI ALT EXEMPLU SIMPLU DE PROGRAM CU VECTORI

C/C++1226

În secțiunea 1225, ați examinat funcțiile membre acceptate de clasa *vector*. În secțiunea 1207 ați creat un program simplu, *vector1.cpp*, care utiliza câteva funcții membre ale clasei *vector* pentru a manevra informațiile dintr-un *vector*. Înainte de a învăța despre clasa *bit_vector* și a o utiliza, vă este util să scrieți un alt porgram ce utilizează funcțiile membre ale clasei *vector* pentru a manevra un *vector*. Următorul program, *vector2.cpp*, utilizează funcțiile membre *reserve*, *max_size*, *resize* și *capacity* cu un *vector* ce conține numai un singur număr întreg:

```
#include <iostream.h>
#include <vector.h>
using namespace std;
typedef vector<int> VECTORINT;
void main(void)
  {
    // Vectorul alocat dinamic are initial 0 elemente.
    VECTORINT Vectorul;
    //Adauga un element la sfarsitul vectorului, un intreg cu
    //valoarea 42.
    Vectorul.push_back(42);
    // Arata statisticile Vectorului.
    cout << "Dimensiunea Vectorului este: " << Vectorul.size()
      << endl;
    cout << "Dimensiunea maxima a Vectorului este: "
     << Vectorul.max_size()<< endl;
    cout << "Capacitatea Vectorului este: " <<
      Vectorul.capacity() << endl;
    // Se asigura ca exista spatiu pentru cel putin 1000 de
    // elemente.
    Vectorul.reserve(1000);
    cout << endl << "Dupa rezervarea de spatiu pentru 1000 de
      elemente:" << endl;
    cout << "Dimensiunea Vectorului este: " << Vectorul.size()
      << endl;
    cout << "Dimensiunea maxima a Vectorului este: " <<
      Vectorul.max_size()<< endl;
    cout << "Capacitatea Vectorului este: " <<
      Vectorul.capacity() << endl;
    // Se asigura ca exista spatiu pentru cel putin 2000 de
    // elemente.
    Vectorul.resize(2000);
    cout << endl << "Dupa rezervarea de spatiu pentru 2000 de
```

```
    elemente " << endl;
  cout << "Dimensiunea Vectorului este: " << Vectorul.size()
    << endl;
  cout << "Dimensiunea maxima a Vectorului este: " <<
    Vectorul.max_size()<< endl;
  cout << "Capacitatea Vectorului este: " <<
    Vectorul.capacity() << endl;
}
```

Programul *vector2.cpp* mai întâi inițializează un *vector*. Apoi, programul adaugă un singur element întreg la *vector* și arată informații despre dimensiunea curentă a vectorului, dimensiunea maximă și capacitatea sa. Apoi, programul rezervă spațiu de stocare pentru 1000 de elemente și afișează aceleași informații despre *vector*. În final, programul redimensionează vectorul pentru a cuprinde 2000 de elemente, apoi afișează informații despre vector. Programul utilizează funcția membră *max_size*, care returnează numărul maxim de elemente pe care vectorul le poate conține precum și funcția membră *capacity*, pentru a returna numărul de elemente pentru care *vector* are alocată memorie în realitate. Atunci când compilați și executați programul *vector2.cpp*, ecranul dumneavoastră va afișa următoarele:

```
Dimensiunea Vectorului este: 1
Dimensiunea maxima a Vectorului este: 1073741823
Capacitatea Vectorului este: 1

Dupa rezervarea de spatiu pentru 1000 de elemente:
Dimensiunea Vectorului este: 1
Dimensiunea maxima a Vectorului este: 1073741823
Capacitatea Vectorului este: 1000

Dupa rezervarea de spatiu pentru 1000 de elemente:
Dimensiunea Vectorului este: 1000
Dimensiunea maxima a Vectorului este: 1073741823
Capacitatea Vectorului este: 1000

Dupa rezervarea de spatiu pentru 2000 de elemente:
Dimensiunea Vectorului este: 2000
Dimensiunea maxima a Vectorului este: 1073741823
Capacitatea Vectorului este: 2000
C:\>
```

1227 O COMPARAȚIE ÎNTRE VECTORI ȘI MATRICELE DIN C



După cum ați văzut, un *vector* este similar unei matrice din C. Totuși există câteva diferențe semnificative între matricele din C și vectori. Ca regulă, vectorii sunt instrumente mult mai utile pentru următoarele motive:

- Programele dumneavoastră pot realoca în mod dinamic dimensiunea unui *vector* la orice punct din execuția programului, așa cum s-a arătat în secțiunea 1226.
- Deoarece clasa *vector* utilizează iteratori, traversarea vectorului și accesarea elementelor conținute este un proces mult mai puternic decât traversarea unei matrice și accesarea elementelor sale. Funcțiile membre *front*, *back*, *begin* și *end*, de exemplu,

vă pun la dispoziție modalități utile de a accesa anumite elemente din *vector*, fără a cunoaște locația exactă a acestor elemente în vector.

- Pe când vectorii alocă o capacitate, matricele din C alocă memorie reală. O matrice de 1000 de întregi din C alocă 2000 de octeți de memorie la declararea sa; un vector ce rezervă 2000 de octeți de memorie dar stochează numai un singur întreg, utilizează numai doi octeți de memorie, după cum ați văzut în secțiunea 1226 (excluzând suprasarcina obiectului – obiectul însuși consumă memorie).
- Dacă introduceți eronat prea multe elemente într-un *vector*, clasa va realoca automat dimensiunea vectorului pentru a cuprinde elementele suplimentare.
- Programele dumneavoastră pot utiliza algoritmi pentru a manevra valorile atât din vectori cât și din matrice. Însă programele dumneavoastră vor putea transfera mai ușor valorile între vectori și alte tipuri ale bibliotecii standard de șabloane, decât între matrice și alte tipuri ale bibliotecii standard de șabloane.

Pe scurt, matricele și vectorii sunt similari. Pe măsură ce programele dumneavoastră continuă să devină mai complexe, asemănarea va crește pentru că veți utiliza vectroii mult mai des decât matricele. Totuși, dacă credeți că sunteți mai acomodat cu matricele din C decât cu vectorii și nu aveți nevoie de flexibilitatea suplimentară a vectorilor, nu vă simțiți obligați să utilizați vectori în locul matricelor.

CONTAINERUL SECVENȚIAL BIT_VECTOR

C/C++1228

În secțiunile de la 1225 până la 1227, ați lucrat mai îndeaproape cu tipul *vector*. Versiunea curentă a bibliotecii standard de șabloane acceptă, de asemenea, tipul *bit_vector*. Un *bit_vector* este în esență, un *vector* de tipul *bool*. Containerul secvențial *bit_vector* are aceeași interfață ca tipul *vector*. Principala diferență dintre un *vector* și un *bit_vector* este aceea că proiectanții bibliotecii standard de șabloane au optimizat clasa *bit_vector* pentru o gestionare mai eficientă a spațiului. Un *vector* va necesita întotdeauna cel puțin un octet pentru fiecare element, pe când un *bit_vector* necesită numai un bit de fiecare element.

Este important să cunoașteți că trebuie să utilizați un iterator de tipul *bit_vector::iterator* cu un *bit_vector*. Motivul pentru modificarea tipului iteratorului este simplu – *bit_vector* utilizează numai un singur bit pentru fiecare element, pe când un vector utilizează minimum un octet. Dacă încercați să utilizați un iterator *vector* cu un *bit_vector*, rezultatul va fi dat de următoarele opt elemente din *bit_vector*, de fiecare dată când incrementați iteratorul. În programele dumneavoastră veți implementa tipul *bit_vector*, după cum arătăm mai jos:

```
bit_vector V(5);
V[0] = true;
V[1] = false;
V[2] = false;
V[3] = true;
V[4] = false;

for (bit_vector::iterator i = V.begin(); i < V.end(); ++i)
  cout << (*i ? '1' : '0');
cout << endl;
```

Observație: *Comitetul de standardizare pentru C++ va elimina tipul **bit_vector** într-o versiune viitoare a bibliotecii standard de șabloane. Motivul pentru care **bit_vector** este o clasă separată și nu o specializare șablon pentru **vector<bool>** este acela că a crea o specializare șablon numai cu tipul **bool** ar impune o specializare parțială a șabloanelor. După ce compilatoarele care acceptă specializarea parțială vor deveni mai răspândite, o specializare pentru **vector<bool>** va înlocui **bit_vector**.*

1229 UTILIZAREA UNUI EXEMPLU SIMPLU CU BIT_VECTOR



După cum ați învățat în secțiunea 1228, un *bit_vector* este un vector pe care îl veți utiliza în special pentru a depozita numai un singur bit pentru fiecare element dintr-un vector. Deoarece *bit_vector* utilizează numai un singur bit, el este capabil să stocheze numai valori de tipul adevărat (*true*) și fals (*false*). Veți utiliza, de asemenea, numai iteratori de tipul *bit_vector::iterator* cu vectorii *bit_vector*. Următorul program, *bvectr1.cpp*, modifică programul *vector1.cpp* pe care l-ați scris în secțiunea 1207 pentru a utiliza un *bit_vector* în locul unui vector de întregi:

```
#include <iostream.h>
#include <bvector.h>

using namespace std;
const DIM_TABLOU = 4;

void main(void)
  {
    // vectorul alocat in mod dimanic nu contine la inceput
    // elemente.
    bit_vector Vectorul(DIM_TABLOU);

    for (int FiecareElem = 0; FiecareElem < DIM_TABLOU;
      FiecareElem++)
      if(FiecareElem>1)
        Vectorul.push_back(FiecareElem - 2);
      else
        Vectorul.push_back(FiecareElem);

    cout << "Primul element: " << Vectorul.front() << endl;
    cout << "Ultimul element: " << Vectorul.back() << endl;
    cout << "Elemente in vector: " << Vectorul.size() << endl;

    // Sterge ultimul element al vectorului. Retineti ca
    // vectorul are baza 0, astfel incat Vectorul.end() indica
    // de fapt cu un element dincolo de capat.
    cout << "Sterge ultimul element." << endl;
    Vectorul.erase(Vectorul.end() - 1);
    cout << "Noul ultim element este: " << Vectorul.back() << endl;

    // Sterge primul element al vectorului.
    cout << "Sterge primul element." << endl;
    Vectorul.erase(Vectorul.begin());
```

```
    cout << "Noul prim element este: " << Vectorul.front() << endl;
    cout << "Elemente in vector: " << Vectorul.size() << endl;
}
```

Atunci când compilaţi şi executaţi programul *bvectr1.cpp*, ecranul dumneavoastră va afişa următorul rezultat:

```
Primul element: 0
Ultimul element: 1
Elemente in vector: 4

Sterge ultimul element.
Noul ultim element este: 0
Sterge primul element.
Noul prim element este: 1
Elemente in vector: 2
C:\>
```

Observaţie: *Nici **Visual C++**, nici **Borland's C++ 5.02** nu cuprind fişierul antet **bvector.h** al clasei **bit_vector** în implementarea lor implicită a bibliotecii standard de şabloane. Dacă doriţi să utilizaţi tipul **bit_vector** în programele dumneavoastră, trebuie să extrageţi fişierul antet dintre fişierele antet ale bibliotecii standard de şabloane şi să îl plasaţi în directorul **include** corespunzător al compilatorului dumneavoastră.*

TIPUL LIST

C/C++ 1230

După cum aţi învăţat, biblioteca standard de şabloane acceptă o varietate de containere. Unul dintre containerele pe care îl veţi utiliza cel mai frecvent şi care ar trebui să vă fie cel mai familiar, este containerul *list*. Un element de tipul *list* este o listă dublu înlănţuită (similară obiectului *lista_inl* pe care l-aţi creat în secţiunile precedente). Aceasta înseamnă că o listă este un container secvenţial care acceptă traversarea înainte şi înapoi şi inserări şi eliminări de elemente de durată lineară, la început, la sfârşit sau în mijloc. Modificările unei liste au următoarele trei proprietăţi importante:

- Inserarea nu invalidează iteratorii către elementele listei.
- Îmbinarea (inserarea unui element în centrul listei) nu invalidează iteratorii către elementele listei.
- Eliminarea elementelor din listă invalidează numai iteratorii care indică elementele eliminate.

Programele dumneavoastră pot schimba ordinea iteratorilor (adică, iteratorul *list<T>::iterator* poate avea după o operaţie pe listă un predecesor sau succesor diferit de cel dinainte), dar clasa nu va invalida iteratorii înşişi şi nu îi va face să indice către elemente diferite, în afara cazului în care invalidarea sau modificarea este explicită.

Notaţi că listele înlănţuite simple, care acceptă numai traversarea înainte, sunt de asemenea utile în anumite cazuri. Dacă nu aveţi nevoie de o parcurgere înapoi, tipul *slist* poate fi mai eficient decât tipul *list*. Veţi învăţa mai multe despre tipul *slist* în secţiunile următoare.

1231 COMPONENTELE GENERICE ALE CONTAINERULUI LIST



După cum ați învățat în secțiunile 1230, biblioteca standard de șabloane acceptă tipul de container *list*. Veți lucra cu tipul *list* al bibliotecii standard de șabloane, tot așa cum ați lucrat cu obiectul *lista_inl* pe care l-ați creat în secțiunile precedente. Cu toate că clasa *list* încapsulează în totalitate echivalentul clasei *obiect_lista*, puteți accesa numai membrii clasei *list* și nu clasa din care derivă.

Clasa *list* poate fi de orice tip. Programele dumneavoastră pot crea liste care acceptă tipuri simple și liste care acceptă tipuri complexe. De exemplu, următoarea definiție creează o listă simplă de întregi, adaugă câteva valori la listă și afișează valorile listei:

```
list<int> L;

L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
   // Afiseaza 1 2 0
```

Codul creează o lista *L*, adaugă trei valori la listă și afișează aceste trei valori. Observați că ultima instrucțiune utilizează algoritmul *copy* al bibliotecii standard de șabloane pentru a copia fiecare element din listă către fluxul de ieșire *cout* și chiar inserează câte un spațiu după fiecare element pe care îl copiază în flux. Tipul *list* acceptă un singur tip generic, *T*. Nu puteți crea liste care să accepte mai multe tipuri într-un singur tip, decât dacă încapsulați mai întâi aceste tipuri într-o clasă sau structură.

1232 CONSTRUIREA UNUI OBIECT DE TIPUL LIST



După cum ați învățat în secțiunea 1230, biblioteca standard de șabloane acceptă tipul de listă dublu înlănțuită, denumit *list*. Deoarece tipul *list* este o clasă, veți inițializa listele cu ajutorul unuia dintre cei câțiva constructori pe care biblioteca standard de șabloane îi definește pentru clasa *list*. Biblioteca standard de șabloane definește patru constructori diferiți pentru clasa *list*, arătați mai jos:

```
explicit list(void);
explicit list(size_type n, const T& val);
list(const list& PrimaLista);
list(const_iterator primul, const_iterator ultimul);
```

Primul constructor desemnează o listă inițial vidă. Al doilea desemnează o repetiție de *n* elemente de valoare *val* (puteți atribui o valoare implicită). Al treilea constructor (constructor de copiere) cere compilatorului să inițializeze noua listă cu o copie a altei liste, *PrimaLista*. Ultimul constructor specifică secvența [primul, ultimul) care sunt doi iteratori dintr-un alt obiect de tipul *list*. Constructorul copiază toate elementele dintre *primul* și *ultimul* din obiectul *list* inițial, în noul obiect *list*. Toți constructorii stochează obiectul alocator, *al*, sau pentru constructorul copie, valoarea returnată de *PrimaLista.get_allocator*, în data membră *allocator*. După stocarea obiectului alocator, toți cei patru constructori inițializează lista.

Pentru a înțelege mai bine diferiții constructori pe care îi prevede clasa *list* a bibliotecii standard de șabloane, studiați următorul program, *list1.cpp*:

```
#include <list.h>
#include <iostream.h>
#include <string.h>

using namespace std;
typedef list<string> LISTSTR;

// testeaza fiecare dintre cei patru constructori
void main(void)
  {
    LISTSTR::iterator i;
    LISTSTR test;    // constructor implicit

    test.insert(test.end(), "unu");
    test.insert(test.end(), "doi");

    LISTSTR test2(test); // construieste dintr-o alta lista
    LISTSTR test3(3, "trei"); // construieste cu trei elemente
            // care contin valoarea "trei"
    LISTSTR test4(++test3.begin(),test3.end());

          // creeaza din parti ale lui test3

    // Afiseaza toate listele
    for (i = test.begin(); i != test.end(); ++i)
      cout << *i << " ";
    cout << endl;

    for (i = test2.begin(); i != test2.end(); ++i)
      cout << *i << " ";
    cout << endl;

    for (i = test3.begin(); i != test3.end(); ++i)
      cout << *i << " ";
    cout << endl;

    for (i = test4.begin(); i != test4.end(); ++i)
      cout << *i << " ";
    cout << endl;
  }
```

Atunci când compilați și executați programul *list1.cpp*, ecranul dumneavoastră va afișa următoarele:

```
unu doi
unu doi
trei trei trei
trei trei
C:\>
```

1233 INSERAREA OBIECTELOR ÎN LISTĂ

C/C++

După cum ați văzut în secțiunea 1232, puteți construi o listă în diverse moduri – unul fără nici o inițializare, două cu inițializare. Mai mult, puteți atribui valori elementelor unei liste în câteva moduri importante. După cum veți învăța în secțiunea 1236, puteți utiliza funcțiile membre *push_back* și *push_front* pentru a adăuga valori obiectului *list*. Totuși, puteți folosi și funcția membră insert pentru a adăuga valori la obiectul *list*. Veți utiliza funcția membră *insert* după cum arătăm în următoarele prototipuri:

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
void insert(iterator it, const_iterator primul,
   const_iterator ultimul);
```

După cum puteți vedea, obiectul *list* pune la dispoziție trei versiuni supraîncărcate ale funcției membre *insert*. Fiecare funcție membră inserează înaintea elementului pe care îl indică iteratorul *it* în secvența controlată, secvența specificată de restul operanziilor. Prima funcție membră inserează un singur element cu valoarea *x* și returnează un iterator care indică noul element inserat. Cea de a doua funcție membră inserează o repetiție de *n* elemente de valoare *x* (cu alte cuvinte, cinci elemente cu valoarea doi, de exemplu). Ultima funcție membră inserează secvența [primul, ultimul) începând de la locația iteratorului *it*.

CD-ROM care însoțește această carte cuprinde programul *insert_3.cpp* care utilizează cele trei metode *insert* ale clasei *list* pentru a insera elemente într-o listă.

1234 UTILIZAREA FUNCȚIEI MEMBRE ASSIGN

C/C++

În secțiunea 1233 ați învățat modul în care propgramele dumneavoastră pot utiliza funcția membră *insert* pentru a insera elemente într-o listă. Programele dumneavoastră pot utiliza, de asemenea, funcția membră *assign* pentru a atribui valori elementelor existente din listă. În loc de inserarea de noi elemente și de eliminarea elementelor vechi, puteți utiliza *assign* pentru a înlocui o serie de elemente dintr-o listă cu ajutorul unei singure instrucțiuni. Veți utiliza funcția membră *assign* sub una dintre cele două forme supraîncărcate ale sale, așa cu arătăm în următoarele prototipuri:

```
void assign(const_iterator primul, const_iterator ultimul);
void assign(size_type n, const T& x = T());
```

Prima funcție membră înlocuiește secvența spre care indică **this* cu secvența [primul, ultimul). Cea de a doua funcție membră înlocuiește secvența spre care indică **this* cu o repetiție de *n* elemente de valoarea *x*. De exemplu, următoarea instrucțiune înlocuiește următoarele patru elemente dintr-o listă cu valoarea 1:

```
AltaLista.assign(4, 1);
```

În funcție de locul din listă spre care indică în mod curent iteratorul, atribuirea poate începe fie la începutul listei, fie în mijlocul ei. Dacă încercați să atribuiți un interval de valori unei serii de elemente ale listei care depășește finalul listei, atribuirea va adăuga elemente suplimentare la capătul listei.

UTILIZAREA FUNCȚIILOR MEMBRE REMOVE ȘI EMPTY

C/C++1235

În secțiunea 1234 ați învățat modul în care programele dumneavoastră pot utiliza funcția membră *assign* pentru a înlocui o serie de elemente dintr-o listă. Atunci când lucrați cu liste, trebuie să eliminați frecvent elemente din listă, în plus față de reatribuirea de valori elementelor din listă. Clasa *list* acceptă două funcții membre supraîncărcate denumite *remove* pe care le puteți utiliza în programele dumneavoastră pentru a elimina elemente dintr-o listă. Veți utiliza cele două versiuni supraîncărcate ale funcției *remove*, după cum arătăm în următoarele prototipuri:

```
iterator remove(iterator it);
iterator remove(iterator primul, iterator ultimul);
```

Prima funcție membră elimină elementul secvenței controlate spre care indică iteratorul *it*. Cea de a doua funcție elimină elementele din intervalul [primul, ultimul) al secvenței controlate. Ambele funcții *remove* vor returna un iterator care desemnează primul element rămas dincolo de elementele eliminate de funcție sau *end* dacă nu există un astfel de element.

Uneori, puteți elimina toate elementele dintr-o listă. Pentru a proteja programele dumneavoastră împotriva încercării de a afișa sau manevra o listă vidă, programele dumneavoastră trebuie să testeze, după eliminări, pentru a determina dacă a mai rămas vreun element în listă. Pentru aceasta, programele dumneavoastră trebuie să utilizeze funcția membră *empty* al cărei prototip este arătat mai jos:

```
bool empty(void) const;
```

Funcția membră *empty* va returna *true* pentru o secvență controlată vidă. Pentru a înțelege mai bine modul în care programele dumneavoastră pot utiliza funcțiile membre *remove* și *empty*, ca și funcția membră *assign*, prezentată în secțiunea 1234, studiați programul *list_are.cpp*, de pe CD-ROM-ul care însoțește această carte, care utilizează toate cele trei funcții. Programul creează două liste, *listOne* și *listAnother*. Apoi programul atribuie trei valori containerului *listOne* și o singură valoare containerului *listAnother*. Apoi, programul utilizează metoda *assign* pentru a copia cele trei elemente din *listOne* în *listAnother* și a suprascrie elementul atribuit în precedenta instrucțiune. După afișarea containerului *listAnother*, programul atribuie valoarea *1* tuturor elementelor listei, ceea ce înlocuiește valorile precedente ale elementelor. Programul afișează din nou containerul *listAnother*, iar apoi șterge primul element din *listAnother* și o afișează din nou. În final, programul șterge toate elementele din *listAnother* și generează mesajul „All gone! (S-au dus toate!)".

TRAVERSAREA OBIECTULUI LIST

C/C++1236

În secțiunea 1235, programul *list_are.cpp* a utilizat funcțiile membre *begin* și *end* împreună cu un iterator *list* pentru a traversa lista. Programele dumneavoastră pot de asemenea să utilizeze funcțiile membre *front și* back pentru a controla poziția unui iterator într-o listă. Funcția membră *front* returnează o referință către primul element al secvenței controlate. Funcția membră *back* returnează o referință către ultimul element al secvenței controlate.

Atunci când programele dumneavoastră utilizează funcțiile *front* și *back*, puteți să utilizați funcțiile *push* și *pop* pentru consistență. Rețineți că funcțiile *push* și *pop* plasează valori într-o

listă la începutul și la finalul ei. Funcția membră *pop_back* elimină ultimul element al secvenței controlate. Funcția membră *pop_front* elimină primul element al secvenței controlate. Toate aceste funcții cer ca secvența controlată să nu fie vidă. Funcția membră *push_front* inserează un element cu valoarea *x* la începutul secvenței controlate. Funcția membră *push_back* inserează un element cu valoarea *x* la sfârșitul secvenței controlate. Pentru a înțelege mai bine activitățile pe care programele dumneavoastră le pot efectua utilizând funcțiile *front, back* și funcțiile de atribuire asociate lor, studiați programul *frntback.cpp*, prezentat mai jos:

```
#include <list.h>
#include <string.h>
#include <iostream.h>

using namespace std;
typedef list<string> LISTSTR;

void main(void)
  {
    LISTSTR test;
    test.push_back("sfarsit");
    test.push_front("mijloc");
    test.push_front("inceput");
    cout << test.front() << endl;  // inceput
    cout << test.back() << endl;   // sfarsit
    test.pop_front();
    test.pop_back();
    cout << test.front() << endl;  // mijloc
  }
```

Atunci când compilați și executați programul *frntback.cpp*, ecranul dumneavoastră va afișa următoarele:

```
inceput
sfarsit
mijloc
C:\>
```

1237 TIPUL SLIST

După cum ați învățat, biblioteca standard de șabloane acceptă tipul *list*, o listă dublu înlănțuită. De asemenea, biblioteca standard de șabloane acceptă și tipul *slist*, care este o listă simplu înlănțuită, care leagă fiecare element de următorul element din listă, dar nu și de elementul precedent. Deci *slist* este un container secvențial ce acceptă traversarea înainte, dar nu și pe cea înapoi, precum și inserări și eliminări de elemente în timp constant. Ca și modificările obiectelor de tip *list*, cele de tip *slist* au următoarele trei proprietăți importante:

- Inserarea nu invalidează iteratorii către elementele listei.
- Îmbinarea (inserarea unui element în centrul listei) nu invalidează iteratorii elementele listei.

- Eliminarea elementelor listei invalidează numai iteratorii care indică către elementele eliminate.

Programele dumneavoastră pot schimba ordinea iteratorilor (adică, iteratorul *slist<T>::iterator* poate avea după o operație pe listă un predecesor sau succesor diferit de cel dinainte), dar modificarea nu va invalida iteratorii înșiși și nu îi va face să indice către elemente diferite, în afara cazului în care invalidarea sau modificarea este explicită.

Principala diferență dintre *slist* și *list* constă în aceea că iteratorii *list* sunt iteratori *bidirectional*, pe când iteratorii *slist* sunt *iteratori forward (de avansare)*. Diferența între tipurile de iteratori semnifică faptul că o listă *list* este mai versatilă decât o listă *slist*. Însă de multe ori iteratorii bidirecționali nu sunt necesari. Ar trebui să utilizați întotdeauna tipul *slist*, în afara cazului în care aveți nevoie de facilitățile suplimentare ale tipului *list*, deoarece listele simplu înlănțuite sunt mai mici și mai rapide decât cele dublu înlănțuite.

Observație: *Ca și în cazul tipului **bit_vector**, nici compilatoarele **Visual C++**, nici **Borland's C++ 5.02** pentru Windows nu cuprind fișierele antet pentru tipul **slist**.*

INSERĂRILE ÎNTR-UN CONTAINER SECVENȚIAL SLIST

C/C++ 1238

În secțiunea 1237 ați învățat despre secvența *slist*. Ca și alte containere secvențiale, *slist* definește funcțiile membre *insert* și *erase*. Însă utilizarea fără atenție a acestor funcții poate avea ca rezultat programe dezastruos de lente. Problema este că primul argument al funcției *insert* este un iterator *pos*, iar *insert* plasează noile elemente după iteratorul *pos*. Cu alte cuvinte, funcția *insert* trebuie să găsească iteratorul exact dinaintea iteratorului *pos*. Găsirea acestui iterator este o operație ce se derulează în timp constant într-o listă *list*, deoarece *list* are iteratori *bidirecționali*. Însă pentru *slist*, funcția *insert* trebuie să traverseze lista de la început până la *pos* pentru a găsi acest iterator. Cu alte cuivnte, *insert* și *erase* sunt operații lente în orice împrejurare, mai puțin pentru zonele apropiate de începutul lui *slist*.

Clasa *slist* pune la dispoziție funcțiile membre *insert_after* și *erase_after*, care sunt operații de durată constantă; ar trebui să utilizați *insert_after* și *erase_after* de fiecare dată când acest lucru este posibil. Dacă credeți că *insert_after* și *erase_after* nu sunt adecvate necesităților dumneavoastră și că trebuie să utilizați mai frecvent *insert* și *erase* la mijlocul listei, probabil că ar fi mai bine să utilizați o listă *list* în loc de *slist*.

CONTAINERUL DEQUE

C/C++ 1239

Un container *deque* este similar unui *vector* deoarece un container *deque* (*coadă*) este un container secvențial ce permite programelor dumneavoastră accesul aleator la elemente și efectuarea de inserări cu durată constantă, ca și eliminări de elemente de la finalul secvenței. O *coadă* de asemenea va permie programelor dumneavoastră, să efectueze inserări și eliminări de elemente de la mijlocul secvenței, de durată lineară.

Principala diferență între o coadă și un vector este aceea că o coadă acceptă inserări și elminări de elemente de la începutul secvenței cu durată constantă (în plus față de suportul vectorilor pentru astfel de activități la finalul secvenței). În plus, o coadă nu are nici o funcție membră similară funcțiilor clasei *vector*, *capacity* și *reserve*, și nu oferă nici una dintre garanțiile de valabilitate a iteratorilor pe care biblioteca standard de șabloane o asociază cu

funcțiile *capacity* și *reserve*. În programele dumneavoastră veți declara și utiliza obiecte de tipul *deque* după cum arătăm mai jos:

```
deque<int> Q;
Q.push_back(3);                    // plaseaza 3 la sfarsit
Q.push_front(1);                   // plaseaza 1 la inceput
Q.insert(Q.begin() + 1, 2 );  // insereaza 2 la mijloc
Q[2] = 0;                          // al treilea element este facut 0
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// valorile care sunt tiparite sunt 1 2 0
```

Funcțiile membre *push_back*, *push_front* și *insert* efectuează asupra unei cozi aceleași operații pe care le efectuează asupra vectorilor. În plus, algoritmul *copy* permite programelor dumneavoastră să copieze ieșirile direct în fluxul de ieșire, tot așa cum ați făcut anterior și cu obiectul *list*.

1240 UTILIZAREA CONTAINERULUI DEQUE



În secțiunea 1239 ați învățat că un container *deque* este similar containerului *vector* și containerului *list*. De fapt, containerul *deque* utilizează elemente ale celorlalte două tipuri de containere. După cum vă așteptați, containerul *deque* acceptă unele funcții membre ca cele pentru vectori și alte funcții membre ca cele pentru stilul liste. Două dintre funcțiile membre pe care containerul *deque* le acceptă sunt funcția membră *swap* (în stilul *vector*) și funcția membră *assign* (în stilul *list*). În programele dumneavoastră veți utiliza funcțiile membre *swap* și *assign* după cum arătăm în următoarele prototipuri:

```
void assign(const_iterator primul, const_iterator ultimul);
void assign(size_type n, const T& x = T());
void swap(deque& dq);
```

Prima funcție membră *assign* înlocuiește secvența către care indică **this* cu secvența [primul,ultimul). Cea de-a doua funcție membră *assign* înlocuiește secvența către care indică **this* cu o repetiție de *n* elemente de valoarea *x*. Funcția membră *swap* interschimbă conținuturile între **this* și *dq*. Pentru a înțelege mai bine prelucrările acestor funcții membre, studiați următorul program, *deque1.cpp*:

```
#include <deque.h>
#include <iostream.h>

using namespace std;
typedef deque<char> CHARDEQUE;

void afis_continut(CHARDEQUE deque, char*);

void main(void)
  {
    CHARDEQUE a(3, 'A');     //creaza a cu 3 A-uri
    CHARDEQUE b(4, 'B');    //creaza b cu 4 B-uri.

    afis_continut(a, "a"); //afiseaza continutul
    afis_continut(b, "b");
```

```
    a.swap(b);          //interschimba a si b
    afis_continut(a, "a");
    afis_continut(b, "b");
    a.swap(b);          // din nou interschimba
    afis_continut(a, "a");
    afis_continut(b, "b");
    a.assign(b.begin(),b.end());  //atribuie continutul lui b la a
    afis_continut(a, "a");
    a.assign(b.begin(),b.begin()+2);
       //atribuie primele doua elemente din b lui a
    afis_continut(a, "a");
    a.assign(3, 'Z');    //atribuie 3 'Z'-uri lui a
    afis_continut(a, "a");
  }
//functie de tiparire a continutului obiectului deque
void afis_continut(CHARDEQUE deque, char *name)
  {
    CHARDEQUE::iterator pdeque;
    cout << "Continutul lui " << name << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
      cout << *pdeque << " ";
    cout<< endl;
  }
```

Atunci când compilaţi şi executaţi programul *deque1.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Continutul lui a : A A A
Continutul lui b : B B B B
Continutul lui a : B B B B
Continutul lui b : A A A
Continutul lui a : A A A
Continutul lui b : B B B B
Continutul lui a : B B B B
Continutul lui a : B B
Continutul lui a : Z Z Z
C:\>
```

UTILIZAREA FUNCŢIILOR MEMBRE ERASE ŞI CLEAR

C/C++1241

În secţiunea 1239 aţi învăţat despre containerul *deque* şi similaritatea sa cu containerul *vector*. În secţiunea 1240 aţi învăţat cum să utilizaţi funcţiile membre *swap* şi *assign* cu un container *deque*. După cum ştiţi, una dintre cele mai obişnuite activităţi pe care le veţi efectua cu o matrice, vector sau altă listă de elemente este de a elimina elemente sau de a elimina valorile elementelor dinăuntrul listei. Containerul *deque* pune la dispoziţie funcţiile membre *erase* şi *clear* pentru a vă ajuta să manevraţi elemente individuale, precum şi întregul container. Funcţia membră *erase* şterge un singur element sau un interval de

elemente. Funcţia membră *clear* şterge toate elementele din coadă. În programele dumneavoastră veţi utiliza aceste funcţii membre după cum arătăm în următoarele prototipuri:

```
iterator erase(iterator iter);
iterator erase(iterator primul, iterator ultimul);
void clear(void) const;
```

Prima funcţie membră *erase* elimină din container elementul către care indică pointerul *iter*. Cea de-a doua funcţie membră *erase* elimină elementele containerului din intervalul [primul,ultimul). Ambele returnează un iterator care desemnează primul element rămas dincolo de elementele eliminate de către funcţiile membre sau *end()* dacă nu mai există nici un asemenea element. Eliminarea a *N* elemente cauzează *N* apeluri ale funcţiilor destructor şi câte o atribuire pentru fiecare element dintre punctul de inserţie şi capătul cel mai apropiat al secvenţei. Eliminarea unui element de la oricare capăt invalidează numai iteratorii şi referinţele care indică elementele scoase. Altfel, ştergerea unui element invalidează toţi iteratorii şi toate referinţele. Funcţia membră *clear* apelează *erase(begin(), end())*. Pentru a înţelege mai bine activităţile pe care aceste funcţii membre le efectuează, studiaţi programul *erase_de.cpp* de pe CD-ROM-ul ce însoţeşte această carte, care utilizează toate cele trei funcţii.

1242 UTILIZAREA OPERATORULUI DE MATRICE [ ] CU O COADĂ

C/C++

După cum aţi învăţat, o coadă (*deque*) este similară unui *vector* şi, prin urmare, similară unei matrice. Deoarece *deque* este similară unei matrice, v-aţi aştepta ca programele dumneavoastră să poată utiliza operatorul de matrice *[]* pentru a accesa elemente specificate dintr-o coadă *deque*. Totuşi, programele pe care le-aţi scris până acum au utilizat iteratori cu obiecte *deque* în loc de un operator de matrice cu un indice. Aşa cum reiese, puteţi utiliza un iterator, operatorul matrice sau pe amândouă, pentru a accesa o *coadă* într-un anumit program. Programele dumneavoastră mai pot să utilizeze şi funcţia membră *at* pentru a trece de la indici de matrice la iteratori.

Funcţia membră *operator[]* returnează o referinţă la elementul secvenţei de pe poziţia *pos*. Dacă această poziţie este incorectă, comportarea funcţiei va fi imprevizibilă. Funcţia membră *at* returnează o referinţă la elementul secvenţei controlate de pe poziţia *pos*. Dacă această poziţie este incorectă, funcţia va lansa un obiect al clasei *out_of_range* ca excepţie. Funcţia membră *empty* returnează *True* pentru o secvenţă controlată vidă. Următorul program, *deq_tab.cpp* utilizează un iterator, operatorul matrice şi funcţia membră *at* pentru a accesa o coadă:

```
#include <deque.h>
#include <iostream.h>

using namespace std;
typedef deque<char> CHARDEQUE;
void afis_continut(CHARDEQUE deque, char*);

void main(void)
  {
    CHARDEQUE a;      // creeaza o coada vida
```

```
    if(a.empty())    // testeaza daca aceasta coada este vida
      cout << "a este vida" << endl;
    else
      cout << "a nu este vida" << endl;
    a.push_back('A');  //insereaza A, B, C si D in a
    a.push_back('B');
    a.push_back('C');
    a.push_back('D');
    if(a.empty())    //testeaza din nou daca a este vida
      cout << "a este vida" << endl;
    else
      cout << "a nu este vida" << endl;
    afis_continut(a,"a"); //afiseaza continutul
    cout << "Primul element al lui a este " << a[0] << endl;
    cout << "Primul element al lui a este " << a.at(0) << endl;
    cout << "Ultimul element al lui a este " << a[a.size()-1]
      << endl;
    cout << "Ultimul element al lui a este " <<
      a.at(a.size()-1) << endl;
  }
//functia de tiparire a continutului unei cozi
void afis_continut(CHARDEQUE deque, char *name)
  {
    CHARDEQUE::iterator pdeque;
    cout << "Continutul lui "<< name << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
      cout << *pdeque <<" ";
    cout << endl;
  }
```

Atunci când compilaţi şi executaţi programul *deq_tab.cpp*, ecranul dumneavoastră va afişa următoarele:

```
a este vida
a nu este vida
Continutul lui a : A B C D
Primul element al lui a este A
Primul element al lui a este A
Ultimul element al lui a este D
Ultimul element al lui a este D
C:\>
```

UTILIZAREA ITERATORILOR INVERŞI CU O COADĂ

C/C++1243

În secţiunile precedente aţi utilizat atât iteratorii, cât şi operatorul matrice pentru a manevra şi a returna valorile dintr-o coadă. Însă cozile acceptă şi iteratorii inverşi. După cum aţi

învățat, un iterator invers este un iterator care indică înapoi, de la finalul cozii către începutul ei. Cu alte cuvinte, atunci când incrementați un iterator invers, el se va mișca în ordine descendentă prin coadă, nu ascendentă ca a iteratorilor de avansare. Figura 1243 arată diferențele dintre mișcarea unui iterator invers și a unui iterator de avansare.

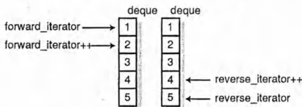


Figura 1243 *Iteratorul invers se mișcă decendent și nu ascendent în cadrul cozii.*

În programele dumneavoastră, puteți utiliza funcțiile membre *rbegin* și *rend* pentru a obține un iterator invers pentru o coadă. Funcția membră *rbegin* returnează un iterator invers care indică exact dincolo de capătul secvenței controlate. Prin urmare, *rbegin* desemnează începutul secvenței inverse. Funcția membră *rend* returnează un iterator invers care indică primul element al secvenței sau imediat dincolo de capătul unei secvențe vide. Prin urmare, *rend* desemnează finalul secvenței inverse. Următorul program, *iter_inv.cpp*, utilizează funcțiile *rbegin* și *rend* pentru a naviga printr-o coadă *deque* în ordine inversă:

```
#include <deque.h>
#include <iostream.h>

using namespace std;
typedef deque<int> INTDEQUE;

void main(void)
  {
   // Creeaza A si o completeaza cu elementele 1,2,3,4 si 5
   // utilizand functia push_back

   INTDEQUE A;
   A.push_back(1);
   A.push_back(2);
   A.push_back(3);
   A.push_back(4);
   A.push_back(5);

   // Acuma ii aifseaza continutul in ordine inversa utilizand
   // reverse_iterator si functiile rbegin() and rend()

   INTDEQUE::reverse_iterator rpi;
   for(rpi = A.rbegin(); rpi != A.rend(); rpi++)
     cout << *rpi << " ";
   cout<< endl;
  }
```

Programul creează coada vidă A, apoi atribuie cozii cinci valori. Apoi, programul defineşte un iterator invers. În final, programul utilizează iteratorul invers pentru a afişa conţinutul cozii în ordine inversă. Atunci când compilaţi şi executaţi programul *iter_inv.cpp*, ecranul dumneavoastră va afişa următoarele:

```
5 4 3 2 1
C:\>
```

MANEVRAREA DIMENSIUNII UNEI COZI

C/C++ 1244

În secţiunile recente, aţi utilizat funcţii membre pentru a crea şi manevra cozi (*deque*). După cum aţi învăţat, clasa *deque* are caracteristici asemănătoare atât clasei *vector* cât şi clasei *list*. Una dintre cele mai puternice facilităţi ale vectorilor, pe care o puteţi utiliza de asemenea şi cu obiecte de tip *deque*, este manevrarea dimensiunii obiectului.Tipul deque pune la dispozitie trei funcţii membre pe care le puteţi utiliza pentru manevrarea dimensiunii unei cozi, dupa cum arătăm mai jos:

```
size_type size(void) const;
void resize(size_type NouaDim, T x = T());
size_type max_size(void) const;
```

Funcţia membră *size* returnează lunigmea (adică numărul curent de elemente) secvenţei. Funcţia membră *resize* modifică dimensiunea la numărul de elemente specificat de parametrul *NouaDim*. Dacă *resize* trebuie să facă secvenţa controlată mai lungă, ea va adăuga elemente cu valoarea *x*. Dacă *resize* nu precizează nici o valoare, valoarea implicită va depinde de tipul obiectelor conţinute de coadă. De exemplu, când coada este o coadă de elemente *char*, elementul implicit va fi un spaţiu alb. Când coada este de întregi, elementul implicit va fi zero. Funcţia membră *max_size* returnează lungimea celei mai mari secvenţe pe care obiectul o poate controla. Pentru a înţelege mai bine activităţile pe care funcţiile membre *size*, *resize* şi *max_size* le efectuează cu obiecte de tipul *deque*, studiaţi programul *deq_size.cpp*, cupirns în CD-ROM-ul care însoţeşte această carte.

Programul creează un obiect *deque* pentru a depozita elemente de tipul *char*. Apoi, programul atribuie patru caractere obiectului *deque*. După ce atribuie cele patru caractere cozii, programul o afişează, îi afişează dimensiunea maximă şi dimensiunea curentă. Apoi programul redimensionează coada la 10 caractere şi completează elementele de la al 5-lea la al 10-lea, cu *X*. Apoi, programul afişează din nou coada şi dimensiunea ei curentă. Apoi programul redimensionează coada la 5 elemente şi tipăreşte aceste elemente. În final, programul tipăreşte dimensiunea curentă şi maximă a obiectului *deque* pentru a arăta că dimensiunea maximă rămâne neschimbată.

OBIECTUL MAP

C/C++ 1245

Un obiect *map* este un container asociativ sortat care asociază obiecte de tipul *Key* cu obiecte de tipul *Data*. Spre deosebire de containerele secvenţiale cu care aţi lucrat anterior, un obiect *map* este un container asociativ pereche (*pair*), aceasta însemnând că tipul valorii sale este *pair<const Key, Data>*. În afară de container asociativ pereche, obiectul *map* este şi un container asociativ unic, ceea ce înseamnă că nu există două elemente cu aceeaşi cheie.

Un obiect *map* vă permite să inserați un nou element în el, fără a invalida iteratorii care indică elementele existente. Dacă ștergeți un element dintr-un set, de asemenea, nu invalidați nici un iterator, exceptând, bineînțeles, acei iteratori care indicau elementul șters.

Obiectele *map* diferă de containerele cu care ați lucrat anterior sub două aspecte: primul, obiectul *map* sortează automat elementele și, al doilea, obiectul *map* utilizează o cheie pe lângă elementul însuși. De exemplu, obiectele *map* sunt utile pentru menținerea unei reprezentări numerice sortate a unei serii de șiruri de caractere, cum ar fi o listă de angajați, organizată după numărul asigurării sociale. Veți lucra cu obiectele *map* în următoarele câteva secțiuni.

1246 UN EXEMPLU SIMPLU DE MAP



După cum ați învățat în secțiunea 1245, programele dumneavoastră pot utiliza obiecte *map* pentru a manevra seturi de obiecte pereche. Pentru a înțelege mai bine modul în care programele dumneavoastră pot utiliza obiectele *map* pentru a manevra informații, studiați următorul program, *map1.cpp*:

```
#include <map.h>
#include <iostream.h>
#include <string.h>
using namespace std;
class ltstr
  {
   public:
     bool operator()(const char* s1, const char* s2) const
       { return (strcmp(s1, s2) < 0);}
  };
void main(void)
{
   map<const char*, int, ltstr> luni;

   luni["Ianuarie"] = 31;
   luni["Februarie"] = 28;
   luni["Martie"] = 31;
   luni["Aprilie"] = 30;
   luni["Mai"] = 31;
   luni["Iunie"] = 30;
   luni["Iulie"] = 31;
   luni["August"] = 31;
   luni["Septembrie"] = 30;
   luni["Octombrie"] = 31;
   luni["Noiembrie"] = 30;
   luni["Decembrie"] = 31;
   cout << "iunie -> " << luni["Iunie"] << endl;
   map<const char*, int, ltstr>::iterator curent =
     luni.find("Iunie");
```

```
   map<const char*, int, ltstr>::iterator prec = curent;
   map<const char*, int, ltstr>::iterator urm = curent;
   ++urm;
   --prec;
   cout << "Precedenta (in ordine alfabetica) este " <<
     (*prec).first << endl;
   cout << "Urmatoarea (in ordine alfabetica) este " <<
     (*urm).first << endl;
}
```

Programul *map1.cpp* creează un obiect *map* al lunilor. Apoi afişează valoarea numerică a lui *iunie* din obiectul *map* sortat (6). Apoi, programul caută în obiectul *map* luna iunie şi afişează pe ecran lunile ale căror nume sunt înainte şi după iunie (adică, în ordine alfabetică, iulie şi martie). Atunci când compilaţi şi executaţi programul *map1.cpp*, ecranul dumneavoastră va afişa următoarele:

```
Iunie -> 6
Precedenta (in ordine alfabetica) este Iulie
Urmatoarea (in ordine alfabetica) este Martie
C:\>
```

UTILIZAREA FUNCŢIILOR MEMBRE PENTRU MANEVRAREA OBIECTELOR MAP

C/C++1247

După cum aţi învăţat în secţiunea 1245, programele dumneavoastră pot utiliza containerele *map* pentru a manevra liste ordonate de obiecte. În secţiunea 1246 aţi utilizat funcţia membră *find* pentru a localiza o valoare într-un obiect *map*. Apoi aţi atribuit doi iteratori la obiectul *map* şi aţi utilizat acei iteratori pentru a afişa valorile din *map*, şi nu valorile cheie. În plus faţă de funcţia membră *find*, programele dumneavoastră mai pot utiliza şi funcţiile membre *end* şi *insert* pentru a manevra informaţii în cadrul contaienrelor de tipul *map*. Veţi utiliza toate aceste trei funcţii după cum arătăm în următoarele prototipuri:

```
// Key este tipul de date dat de primul argument al definitiei
// pentru map
iterator map::find(const Key& cheie);
iterator map::end(void);
pair<iterator, bool> map::insert(const value_type& x);
```

Funcţia *end* returnează un iterator indicând o poziţie dincolo de finalul unei secvenţe. Funcţia *find* returnează un iterator ce desemnează primul element a cărui cheie de sortare este egală cu parametrul *cheie*. Dacă nu există un astfel de element, iteratorul va fi cel returnat de *end()*. Dacă cheia nu există încă, *insert* o va adăuga secvenţei şi va returna *pair<iterator, true>*. Dacă cheia există deja, *insert* nu va adăuga cheia secvenţei şi în schimb va returna *pair<iterator, false>*. Următorul program, *map_ints.cpp* creează un obiect de tipul *map* de elemente întregi şi şiruri de caractere. În acest caz, codificarea este cea dintre cifre şi denumirea lor sub formă de şir de caractere (1 devine „Unu", 2 devine „Doi" şi aşa mai departe). Programul citeşte un număr dat de către utilizator, găseşte cuvântul echivalent pentru fiecare cifră (cu ajutorul obiectului *map*) şi tipăreşte numerele ca şiruri de caractere. De exemplu, dacă utilizatorul introduce 25463, programul va răspunde cu: *Doi Cinci Patru*

Sase Trei. Programul *map_ints.cpp*, arătat mai jos, utilizează funcțiile membre *find*, *end* și *insert*:

```
#include <map.h>
#include <iostream.h>
#include <string.h>

using namespace std;
typedef map<int, string, less<int>> INT2STRING;

void main(void)
  {
    // Creeaza un obiect map de intregi si siruri de caractere
    INT2STRING obMap;
    INT2STRING::iterator Iteratorul;
    string Sirul = "";
    int index, continue = 1;

    // Completeaza obMap cu cifrele de la 0 la 9, fiecare
    // codificata
    // cu sirul de caractere corespunzator
    // Observatie: value_type este o pereche(pair) pentru
    // obiectele map
    obMap.insert(INT2STRING::value_type(0,"Zero"));
    obMap.insert(INT2STRING::value_type(1,"Unu"));
    obMap.insert(INT2STRING::value_type(2,"Doi"));
    obMap.insert(INT2STRING::value_type(3,"Trei"));
    obMap.insert(INT2STRING::value_type(4,"Patru"));
    obMap.insert(INT2STRING::value_type(5,"Cinci"));
    obMap.insert(INT2STRING::value_type(6,"Sase"));
    obMap.insert(INT2STRING::value_type(7,"Sapte"));
    obMap.insert(INT2STRING::value_type(8,"Opt"));
    obMap.insert(INT2STRING::value_type(9,"Noua"));
    // Citeste un numar de la tastatura si il afiseaza ca cuvinte
    while (continua)
      {
       cout << "Introduceti \"q\" pentru a iesi sau introduceti
         un numar: ";
       cin >> Sirul;
       if(Sirul == "q")
         continua = 0;
       // extrage fiecare cifra din sir, gaseste intrarea
       //corespunzatoare din map(cuvantul echivalent) si/l afiseaza
       for( index = 0; index < Sirul.length(); index++)
         {
           Iteratorul = obMap.find(Sirul[index] - '0');
           if(Iteratorul != obMap.end())  // este 0 - 9
             cout << (*Iteratorul).second << " ";
           else       // nu este 0 - 9
```

```
            cout << "[eroare] ";
        }
        cout << endl;
    }
}
```

Programul creează un obiect simplu de tipul *map* de numere şi obiecte *string*. Atunci când întâlneşte un număr din *map*, programul va afişa echivalentul în şi r de caractere al numărului (de exemplu 21 produce ieşirea *Doi Unu*). Programul creează obiectul de tip map, apoi introduce într-o buclă infinită până când utilizatorul selectează tasta q – punct în care programul se opreşte. Atunci când compilaţi şi executaţi programul *map_ints.cpp*, ecranul dumneavoastră va afişa următoarele (presupunând că aţi introdus aceleaşi numere):

```
Introduceti "q" pentru a iesi sau introduceti un numar: 911
Noua Unu Unu
Introduceti "q" pentru a iesi sau introduceti un numar: 1500
Unu Cinci Zero Zero
Introduceti "q" pentru a iesi sau introduceti un numar: 4995
Patru Noua Noua Cinci
Introduceti "q" pentru a iesi sau introduceti un numar: q
[eroare]
C:\>
```

CONTROLAREA DIMENSIUNII ŞI CONŢINUTULUI UNUI OBIECT MAP

C/C++1248

În secţiunea 1247, aţi utilizat funcţia *insert* pentru a adăuga zece perechi de valori unui obiect *map*. Pe măsură ce programele dumneavoastră devin mai complexe, uneori va trebui să eliminaţi sau să înlocuiţi elemente dintr-un obiect *map*, iar uneori veţi avea nevoie să determinaţi cea mai mare dimensiune posibilă a obiectului *map*. Implementarea tipului *map* din biblioteca standard de şabloane vă permite să efectuaţi toate aceste acţiuni asupra obiectelor dumneavoastră de tipul *map*. De fapt, cele patru funcţii membre pe care le veţi utiliza pentru a efectua aceste activităţi sunt similare celor pe care le-aţi utilizat anterior cu alte obiecte ale bibliotecii standard de şabloane, după cum arătăm mai jos:

```
size_type max_size(void) const;
void clear(void) const;
bool empty(void) const;
iterator erase(iterator primul, iterator ultimul);
```

Funcţia *max_size* vă permite să stabiliţi limita superioară a dimensiunii unui obiect de tipul *map*. Funcţia *clear* vă permite să eliminaţi toate elementele dintr-un obiect *map*. Asemănător, funcţia *empty* permite programelor dumneavoastră să determine dacă un obiect *map* mai conţine sau nu elemente. În fine, funcţia *erase* vă permite să eliminaţi un interval de elemente dintr-un obiect *map*. (Reţineţi că utilizarea instrucţiunii *clear* este echivalentă cu apelul *[erase(map.begin, map.end])* – utilizarea lui *clear* doar face mai evident care este acţiunea instrucţiunii). Pentru a înţelege mai bine activităţile pe care le efectuează aceste funcţii membre, studiaţi programul *map_mon2.cpp* aflat pe CD-ROM-ul ce însoţeşte această carte, care creează din nou un obiect *map* cu luni. Însă programul *map_mon2.cpp* şterge apoi obiectul *map* şi îl completează din nou cu un obiect *map* reprezentând numele zilelor săptămânii. Programul *map_mon2.cpp* creează un obiect *map* de întregi şi şiruri de caractere

și îl completează mai întâi cu un *map* de nume ale lunilor asociate numerelor lunilor. Apoi programul șterge și recompletează obiectul cu un obiect *map* reprezentând numele zilelor săptămânii asociate numerelor întregi corespunzătoare.

1249 TIPUL SET

C/C++

Un *set* este un container asociativ ordonat care depozitează obiecte de tipul *key*. În plus față de capacitatea sa de a fi ordonat, un *set* este și un container asociativ simplu, ceea ce înseamnă că tipul valorii sale, ca și tipul cheii, este același, *key*. În fine, un set este și un container asociativ unic, ceea ce înseamnă că nu există două elemente identice în container. Un set este în esență un obiect de tipul *map* având un singur tip și nu un tip valoare și un alt tip cheie. După cum ați învățat, cheile sunt constante și nu se pot modifica. Atunci când lucrați cu *seturi* fiecare element este nemodificabil după ce i-ați atribuit o valoare inițială. Programele dumneavoastră vor trebui să șteargă valoarea și să adauge la set un element de înlocuire.

Tipurile *set* și *multiset* sunt în particular foarte conevenabile pentru algoritmii de lucru cu seturile conținute în biblioteca standard de șabloane: *set_union* (reuniune), *set_intersection* (intersecție), *set_difference* (diferență) și *set_symmetric_difference* (diferența simetrică). Motivul este dublu: în primul rând, algoritmii cu seturi cer ca argumentele să fie intervale ordonate și, deoarece *set* și *multiset* sunt containere asociative ordonate, fiecare container își sortează elementele sale în ordine ascendentă. În al doilea rând, intervalul de ieșire al acestor algoritmi este întotdeauna sortat și inserarea unui interval sortat într-un element de tipul *set* sau *multiset* este o operațiune rapidă: implementările containerelor asociative ordonate unice și multiple garantează că inserarea unui interval se va desfășura cu o durată lineară dacă intervalul este de asemena ordonat.

Un *set* vă permite inserarea unui nou element într-un set fără a invalida iteratorii care indică spre elementele existente. Un *set* vă permite de asemenea să ștergeți un element din el fără a invalida vreun iterator, cu excepția, bineînțeles, a iteratorilor care indicau către elementul eliminat. Următorul program, *set1.cpp*, creează două seturi simple, compară valorile lor și creează un al treilea set, după cum arătăm mai jos:

```
#include <set.h>
#include <iostream.h>

using namespace std;
class ltstr
   {
     public:
       bool operator()(const char* s1, const char* s2) const
         { return (strcmp(s1, s2) < 0);}
   };

void main(void)
   {
     const int N = 6;
     const char* a[N] = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
     const char* b[N] = {"ggg", "hhh", "eee", "iii", "ccc", "aaa"};
     set<const char*, ltstr> A(a, a + N);
```

```
    set<const char*, ltstr> B(b, b + N);
    set<const char*, ltstr> C;
    cout << "Setul A: ";
    copy(A.begin(), A.end(), ostream_iterator<const
      char*>(cout, " "));
    cout << endl;
    cout << "Setul B: ";
    copy(B.begin(), B.end(), ostream_iterator<const
      char*>(cout, " "));
    cout << endl;
    cout << "Reuniunea: ";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
         ostream_iterator<const char*>(cout, " "), ltstr());
    cout << endl;
    cout << "Intersectia: ";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(),
      ostream_iterator<const char*>(cout, " "), ltstr());
    cout << endl;
    set_difference(A.begin(), A.end(), B.begin(),
      B.end(),inserter(C, C.begin()), ltstr());
    cout << "Setul C (diferenta dintre A si B): ";
    copy(C.begin(), C.end(), ostream_iterator<const
      char*>(cout, " "));
    cout << endl;
 }
```

Programul *set1.cpp* creează un set reuniune, un set intersecţie şi un set diferenţă. Setul reuniune conţine toate valorile din ambele seturi iniţiale, în ordine crescătoare. Intersecţia conţine toate valorile conţinute atât în setul A, cât şi în setul B, dar nu şi valorile ce nu sunt conţinute în ambele seturi. În fine, diferenţa păstrează valorile care apar în primul set, dar nu şi în cel de-al doilea. Atunci când compilaţi şi executaţi programul, ecranul dumneavoastră va afişa următoarele:

```
Setul A: aaa bbb ccc ddd eee fff
Setul B: aaa ccc eee ggg hhh iii
Reuniunea: aaa bbb ccc ddd eee fff ggg hhh iii
Intersectia: aaa ccc eee
Setul C (diferenta dintre A si B): bbb ddd fff
C:\>
```

UN EXEMPLU SIMPLU DE SET

C/C++1250

În secţiunea 1249 aţi învăţat despre obiectele *set* şi aţi scris un program simplu, *set1.cpp*, care utiliza tipul *set* şi genera ieşiri pe ecran. Cum nu avem la dispoziţie suficient spaţiu pentru a anliza tipul *set* în detaliu, această secţiune vă va prezenta pe scurt funcţiile *lower_bound*, *upper_bound* şi *equal_range*, funcţii pe care programele dumneavoastră le pot utiliza cu tipul *set*.

Funcţia *lower_bound* returnează un iterator către cel mai apropiat element din secvenţa controlată care are o cheie diferită de valoarea pe care programul a transmis-o funcţiei

lower_bound. Funcția *upper_bound* returnează un iterator către cel mai apropiat element din secvența controlată care are o cheie egală cu valoarea pe care programul a transmis-o funcției *upper_bound*. Dacă nu există un astfel de element, funcția returnează *end*. În ambele cazuri, programul utilizează funcția *set::key_comp(key, x)* pentru a determina dacă cheia se potrivește. Funcția *equal_range* returnează o valoare pereche (*pair*), unde *.first* este rezultatul funcției *lower_bound*, iar *.second* rezultatul funcției *upper_bound*. Pentru a înțelege mai bine activitățile pe care le efectuează funcțiile membre ale clasei *set*, studiați următorul program, *set_rang.cpp*:

```
#include <set.h>
#include <iostream.h>

using namespace std;
typedef set<int, less<int>> SET_INT;

void main(void)
  {
    SET_INT s1;
    SET_INT::iterator i;

    s1.insert(5);
    s1.insert(10);
    s1.insert(15);
    s1.insert(20);
    s1.insert(25);
    cout << "s1 - incepand la s1.lower_bound(12)" << endl;

// afiseaza: 15,20,25
    for (i = s1.lower_bound(12); i != s1.end(); i++)
      cout << "s1 are " << *i << " in setul sau." << endl;
    cout << "s1 - incepand la s1.lower_bound(15)" << endl;

// afiseaza: 15,20,25
    for (i = s1.lower_bound(15);i != s1.end(); i++)
      cout << "s1 are " << *i << " in setul sau." << endl;
    cout << "s1 - incepand la s1.upper_bound(12)" << endl;

// afiseaza: 15,20,25
    for (i = s1.upper_bound(12); i != s1.end(); i++)
      cout << "s1 are " << *i << " in setul sau." << endl;
    cout << "s1 -- incepand la s1.upper_bound(15)" << endl;

// afiseaza: 20,25
    for (i = s1.upper_bound(15); i != s1.end(); i++)
      cout << "s1 are " << *i << " in setul sau." << endl;
    cout << "s1 -- s1.equal_range(12)" << endl;

// nu afiseaza nimic
    for (i = s1.equal_range(12).first;i !=
      s1.equal_range(12).second; i++)
      cout << "s1 are " << *i << " in setul sau." << endl;
    cout << "s1 -- s1.equal_range(15)" << endl;
```

```
// afiseaza: 15
  for (i = s1.equal_range(15).first;i !=
     s1.equal_range(15).second; i++)
    cout << "s1 are " << *i << " in setul sau." << endl;
 }
```

Programul *set_rang.cpp* arată cum se utilizează funcția *lower_bound* pentru a obține un iterator către cel mai apropiat element din secvența controlată care are o cheie ce nu se potrivește valorii transmise funcției. De asemenea, acest program arată cum se utilizează funcția *upper_bound* pentru a obține un iterator către cel mai apropiat element din secvența controlată care are o cheie ce se potrivește valorii transmise funcției. Ultimul lucru pe care ni-l arată acest program este modul de utilizare al funcției *equal_range* epntru a obține o valoare pereche (*pair*) ce conține rezultatele *lower_bound* și *upper_bound* ale cheii. Atunci când compilați și executați programul *set_rang.cpp*, ecranul dumneavoastră va afișa următoarele:

```
s1 -- incepand la s1.lower_bound(12)
s1 are 15 in setul sau.
s1 are 20 in setul sau.
s1 are 25 in setul sau.
s1 -- incepand la s1.lower_bound(15)
s1 are 15 in setul sau.
s1 are 20 in setul sau.
s1 are 25 in setul sau.
s1 -- incepand la s1.upper_bound(12)
s1 are 15 in setul sau.
s1 are 20 in setul sau.
s1 are 25 in setul sau.
s1 -- incepand la s1.upper_bound(15)
s1 are 20 in setul sau.
s1 are 25 in setul sau.
s1 -- s1.equal_range(12)
s1 -- s1.equal_range(15)
s1 are 15 in setul sau.
C:\>
```

INTRODUCERE ÎN PROGRAMAREA WIN32

C/C++1251

În primele 1250 de secțiuni ați învățat programarea în C și C++. Cu toate că programele scrise în secțiunile precedente pot rula sub Windows, ele au fost concepute pentru mediile DOS sau UNIX. În următoarele 250 de secțiuni veți învăța fundamentele programării sub Windows. Multe din secțiunile care urmează vor relua elemente scrise în programele precedente cu explicații privind prelucrarea lor similară utilizând interfața Windows cu programul de aplicații pe care cartea de față și multe alte cărți de programare în Windows o denumesc *Win32 API*. Este important să înțelegem că această carte se referă la Win32 API și nu la Win16 API, ceea ce înseamnă că apelarea funcțiilor prezentate în această carte vor funcționa sub Windows 95 și Windows NT, nu și sub Windows 3.11 sau versiuni mai vechi.

În plus, datorită diferențelor dintre cele două sisteme de operare, vor exista situații în care secțiunea va prezenta o funcție API pe care o va accepta fie numai Windows 95, fie numai Windows NT. De regulă aceste secțiuni vor atenționa ce sisteme de operare vor accepta

funcția API respectivă și de ce. Dacă aveți probleme cu programul cuprins într-o anumită secțiune trebuie să reverificați programul pentru a vă asigura că acel cod este compatibil cu sistemul de operare.

În fine, așa cum probabil cunoașteți, ieșirea standard generată de Windows este foarte diferită de cea generată până acum. Un simplu program DOS care va genera ieșirea „Jamsa's C/C++ Programmer's Bible" va arăta așa:

```
Jamsa's C/C++ Programmer's Bible
C:\>
```

În schimb, un program Windows simplu care produce o casetă de mesaj cu o informație similară va genera o ieșire cum este cea din Figura 1251

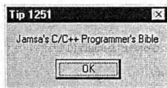


Figura 1251 *O casetă de mesaj simplă sub Windows.*

1252 ALTE DIFERENȚE ÎNTRE PROGRAMELE DOS ȘI WINDOWS

C/C++

Așa cum ați văzut, în secțiunea 1251, există diferențe semnificative între aspectul ieșirii programelor generate de Windows și al celor generate de programele DOS. Însă, diferențele între programarea DOS și Windows sunt mult mai nete decât caracteristicile de ieșire, deși acestea sunt mai ușor de înțeles. Lista de mai jos descrie în detaliu o parte din diferențele dintre programarea în Windows și în DOS:

- Deoarece Windows este un sistem de operare multitasking (în care două sau mai multe programe pot rula în același timp), programele vor partaja spațiul în memorie cu cel puțin un alt program, dar cel mai adesea, cu multe alte programe. Este foarte important să vă asigurați că programele rămân în spațiul de memorie alocat și că nu modifică spațiul de memorie al celorlalte programe.
- Deoarece Windows prelucrează informații bazate pe text în mod diferit față de DOS, nu veți mai genera ieșiri cu *printf* sau *cout*. Veți utiliza, însă, frecvent, matrice flux pentru formatarea informațiilor, înainte de trimiterea lor într-o fereastră.
- Deoarece interfața grafică Windows poate primi intrări de la utilizator teoretic în orice punct al prelucrării programului, programele Windows, în general, nu vor fi la fel de liniare ca programele DOS. Cu alte cuvinte, veți construi programele Windows în contextul răspunsului la acțiunile utilizatorului și al mesajelor sistemului.
- Programele Windows vor consta, în general, din mai multe clase în cadrul mai multor fișiere clasă și antet și sunt deseori semnificativ mai extinse decât programele DOS, deoarece programele Windows nu sunt limitate la memoria convențională.

- Programele Windows pe care le proiectați în această carte, sunt, în general, mai apte să primească intrări de la utilizator decât multe dintre programele DOS proiectate în secțiunile precedente. De regulă, datorită faptului că specificul grafic al mediului Windows necesită interacțiunea cu utilizatorul în vederea unor prelucrări eficiente, majoritatea programelor scrise pentru Windows vor asigura suport pentru o serie de interacțiuni cu utilizatorul.
- În ciuda tuturor diferențelor prezentate în această listă și a celor pe care le veți învăța în următoarele secțiuni, este important să rețineți că programarea C++ în Windows este fundamental aceeași cu programarea C++ efectuată până acum, multe dintre conceptele aplicate fiind puțin diferite de cele din DOS.

În multe dintre secțiunile următoare veți învăța bazele programării Windows și impactul lor asupra programelor. Veți începe să scrieți programe în Windows și, în momentul terminării cărții de față, veți fi pregătiți să creați programe Windows care gestionează memoria, fișierele, grafica, imprimantele și altele.

Observație: *Compilatorul Turbo C++ Lite care însoțește cartea de față nu este proiectat să scrie programe care rulează sub Windows. Pentru a scrie programe sub Windows trebuie să dețineți un compilator dedicat pentru Windows, cum este Microsoft Visual C++ 5.0 sau Borland C++ 5.02. Programele prezentate în următoarele secțiuni vor opera sub unul dintre aceste compilatoare.*

PREZENTAREA FIRELOR DE EXECUȚIE (THREADS)

C/C++1253

După cum ați aflat din parcurgerea capitolelor pentru DOS ale acestei cărți, ca o regulă generală, DOS nu acceptă decât executarea unui singur program în memorie la un moment dat. Așa cum ați aflat deja, Windows nu se supune acestei constrângeri. În realitate, Windows limitează numărul de programe pe care le poate executa calculatorul la un moment dat. Dacă este suficientă memorie pentru a deschide programe suplimentare, Windows gestionează toate aceste programe în memorie folosind firele de execuție (*threads*). Simplu spus, un fir de execuție este o modalitate de a vizualiza cerințele unui program de utilizare a microprocesorului (CPU). Când o multitudine de programe concurează la executare, sistemul de operare va gestiona unul sau mai multe fire de execuție pentru fiecare program. Windows plasează fiecare fir de execuție într-o coadă de fire (o listă). Mai târziu, în funcție de prioritatea acordată firelor, Windows va extrage un fir din listă (*thread queue*) și îl va aloca microprocesorului, care la rându-i va executa instrucțiunile firului. O prioritate de fir (*thread priority*) determină dacă Windows plasează un fir în fața listei sau după alte fire. Figura 1253 arată o simplă distribuire grafică a unei scheme multitasking a firelor.

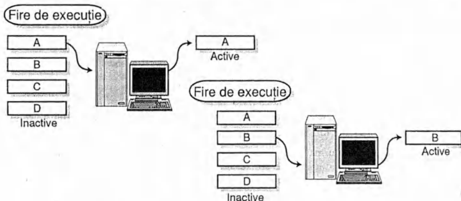


Figura 1253.1 *CPU prelucrează o serie de fire.*

Veți învăța în detaliu mai multe despre fire în secțiunile următoare. Pentru moment însă, să înțelegem că firul este entitatea pe care Windows o atribuie microprocesorului pentru a executa instrucțiunile din program. Windows ordonează firele în funcție de prioritatea lor, iar CPU le execută pe fiecare în parte. Figura 1253.2 arată un model simplu despre modul în care programul solicită procesări de la CPU și primește informații când CPU termină procesarea.

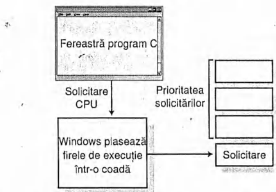


Figura 1253.2 *Un model simplu al schemei de prelucrare a firelor în Windows.*

Așa cum veți învăța în secțiunea 1254, Windows returnează informații către program sub formă de mesaje, de fiecare dată când prelucrează un fir.

1254 MESAJELE



Mijlocul fundamental de comunicare utilizat de Windows și programele scrise sub Windows este *mesajul*. Pe scurt, de fiecare dată când apare o operație, Windows răspunde la acea operație sau acțiune printr-un mesaj către sine sau către alt program. De exemplu, când un

utilizator execută un clic de mouse în fereastra programului, Windows citește acel clic de mouse și trimite un mesaj către program care precizează că utilizatorul a executat un clic de mouse în fereastra programului la o anumită locație. Recepționând acest mesaj, programul va începe propria sa prelucrare ca răspuns la acest mesaj. Dacă mesajul nu este important pentru program, el va fi pur și simplu ignorat. Pentru a înțelege mai bine modelul de mesaje din Windows, analizați figura 1254 care prezintă modelul într-o formă liniară simplificată.

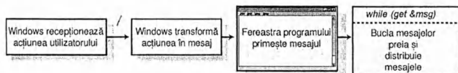


Figura 1254 *O diagramă simplă a modelului de mesaje în Windows.*

Când scrieți programe sub Windows, cele mai importante rutine din program vor fi cele care acceptă și prelucrează mesajele de la sistemul de operare. De exemplu, următorul fragment de cod al funcției *WndProc* arată o funcție de prelucrare simplă a mesajului și a răspunsului:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
 {
   switch(uMsg)
    {
     case WM_COMMAND:
      switch(LOWORD(wParam))
       {
        case IDM_TEST:
         break;
          case IDM_EXIT:
           DestroyWindow(hWnd);
           break;
       }
      break;
     case WM_DESTROY:
      PostQuitMessage(0);
      break;
     default:
      return(DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
   return(0L);
 }
```

În general fragmentul de cod folosește instrucțiunea *switch* pentru a determina mărimea mesajului. O dată cu parcurgerea secțiunilor care urmează veți înțelege repede procesul executat în cadrul funcției *WndProc.* În fragmentul anterior, funcția testează mesajul *WM_COMMAND* sau *WM_DESTROY* sau transferă funcția programului handle implicit pentru mesaje. În funcție de mesajul primit, funcția execută prelucrarea corespunzătoare.

1255 COMPONENTELE WINDOWS



După cum aţi bănuit, blocul de construcţie al tuturor programelor Windows îl reprezintă una sau mai multe ferestre sau casete de dialog (un tip special de fereastră). Concret, aproape fiecare obiect dintr-o fereastră este la rândul lui o fereastră de un anumit tip. În secţiunile următoare veţi înţelege mai bine că toate ferestrele folosite de programul dumneavoastră sunt similare, dar derivate din locaţii diferite. În cazul nostru, este important să înţelegem componentele unei „ferestre standard" – cu alte cuvinte, ceea ce va înţelege utilizatorul dumneavoastră prin fereastră.

Windows îşi construieşte în general „fereastra standard" din şapte piese de bază. Puteţi fragmenta aceste piese mai departe, după cum veţi face în următoarele secţiuni. Este util însă să înţelegem alcătuirea de ansamblu a ferestrei înainte de a analiza fiecare din piesele mai mici care alcătuiesc o singură componentă de fereastră. Cele şapte componente de bază ale ferestrei sunt prezentate în următoarea listă:

- *Cadrul fereastră* (*window frame*) este containerul tuturor componentelor dintr-o fereastră. Aşa cum veţi învăţa puteţi face mai multe tipuri de cadre fereastră. Cel mai obişnuit cadru fereastră este fereastra redimensionabilă ca cea din Figura 1255.1. În programul dumneavoastră veţi manipula cadrul şi veţi primi numeroase mesaje (cum sunt mesajele de redimensionare) de la fereastra cadru. Secţiunile următoare vor prezenta în detaliu cadrul fereastră.
- *Bara de titlu* (*title bar*) oferă utilizatorului informaţii despre program. Bara de titlu extinde dimensiunea ferestrei de-a lungul laturii de sus. Bara de titlu identifică ce arată fereastra şi permite utilizatorului să execute numeroase operaţii cu fereastra. Bara de titlu este punctul de control pentru deplasarea ferestrei şi este locul de plasarea a meniului sistem cu butoanele *MINIMIZE, MAXIMIZE, RESTORE* şi *CLOSE WINDOW*. Aşa cum veţi învăţa în secţiunile următoare, compoziţia barei de titlu va diferi mult în funcţie de aplicaţiile care produc ferestre şi de scopul ferestrelor în care sunt plasate barele de titlu.
- Butoanele *MINIMIZE, MAXIMIZE, RESTORE* şi *CLOSE WINDOW* plasate în containerul barei de titlu sunt suficient de importante pentru a le considera componente ale ferestrei înseşi. Butoanele vă permit controlul dimensiunilor ferestrei şi să închideţi fereastra când terminaţi.
- *Suprafaţa client* (*client area*) este suprafaţa din cadrul ferestrei pe care programul dumneavoastră o poate personaliza şi ar trebui să o proiecteze cu intenţia de a primi intrări de la utilizator. Cu alte cuvinte, suprafaţa client este secţiunea ferestrei unde se desfăşoară majoritatea acţiunilor. De exemplu, dacă lucrăm cu un document Microsoft Word, suprafaţa client este regiunea în care introducem şi edităm documentul în cadrul ferestrei.
- *Bara de defilare* (*scroll bar*) permite utilizatorului să navigheze de la dreapta la stânga şi de sus în jos în interiorul ferestrei. De exemplu, programul dumneavoastră va utiliza barele de defilare pentru a permite utilizatorului să vadă mai mult dintr-un formular de completat. De asemenea, programul dumneavoastră poate utiliza barele de defilare pentru a permite utilizatorului să se deplaseze într-un document salvat anterior pe disc (la fel ca folosirea barelor de defilare din Microsoft Word). Barele de defilare sunt un instrument de navigare foarte important pentru programele Windows. Un

echivalent apropiat din programele DOS sunt tastele cu săgeți, în funcție de modul cum sunt programate într-o aplicație dată.

- *Bara de meniu* (*menu bar*) este o componentă a majorității ferestrelor părinte, dar de regulă nu sunt prezente în multe din ferestrele copil (*child windows*). Secțiunea 1256 prezintă ferestrele părinte și copil în detaliu. Programul din Windows utilizează bara de meniu pentru a oferi utilizatorului opțiuni corespunzătoare programului. Așa cum veți învăța, fiecare program Windows cuprinde cel puțin opțiunile de meniu File și Help. Programele Windows mai complexe pot cuprinde 10 sau mai multe meniuri și fiecare meniu poate conține 20 sau mai multe opțiuni. Pe măsură ce programul dumneavoastră devine mai complex, meniurile vor deveni și ele tot mai complexe. Figura 1255.1 arată bara de meniu a programului Microsoft Word

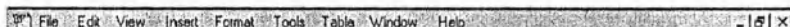


Figura 1255.1 *Bara de meniu a programului Microsoft Word.*

- *Bara de stare* (*status bar*) este o componentă a majorității ferestrelor dar de regulă nu este prezentă la ferestrele copil. Programele din Windows folosesc bara de stare pentru a oferi utilizatorului informații specifice cu detalii despre poziția curentă în program – fie ea o poziție într-un document, o linie etc. Figura 1255.2 arată bara de stare a produsului Microsoft Word care furnizează utilizatorului informații despre poziția curentă în cadrul documentului curent.

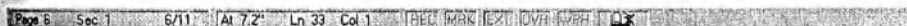


Figura 1255.2 *Bara de stare a produsului Microsoft Word.*

PREZENTAREA FERESTRELOR PĂRINTE ȘI FERESTRELOR COPIL

C/C++ 1256

În secțiunea 1255 ați învățat că majoritatea ferestrelor părinte cuprind o bară de meniu și o bară de stare, în timp de ferestrele copil nu. Poate nu înțelegeți încă ce sunt ferestrele părinte și ferestrele copil. Majoritatea programelor din Windows furnizează interfața de document multiplu (MDI – *Multiple Document Interface*) care permite programului să mențină și să afișeze componente multiple într-o singură fereastră. De exemplu, Figura 1256.1 arată Microsoft Word cu patru ferestre cu documente deschise.

La fel ca ierarhia claselor despre care ați învățat și le-ați folosit în multe programe C++, fiecare fereastră în Windows derivă dintr-o fereastră de bază. Astfel, fiecare fereastră are o fereastră părinte. În Figura 1256.1 ferestrele copil sunt ferestrele document interne, iar fereastra părinte este fereastra Microsoft Word. Ați putea să considerați, totuși, că la rândul ei fereastra Microsoft Word este fereastră copil față de Windows Desktop care îi este părinte. Windows Desktop nu are o fereastră părinte.

În figura precedentă, fereastra părinte conține o interfață multiplu-document, care permite ferestrei părinte să dețină mai multe ferestre copil. Alte programe Windows conțin interfața cu un singur document care permite ferestrei părinte să dețină numai o fereastră copil, ca în figura 1256.2 de mai jos.

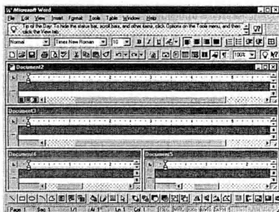


Figura 1256.1 *Fereastra părinte Microsoft Word cu patru ferestre copil deschise.*



Figura 1256.2 *Programul Notepad din Windows este o interfață cu un singur document.*

În sfârșit, multe programe recente din Windows conțin o varietate specială de interfață cu un singur document, denumită curent interfață document de tip *Explorer*. Programatorii au dat această denumire după modelul de interfață al programului *Windows Explorer*. Interfața document de tip *Explorer* este similară în toate privințele interfeței cu un singur document, cu excepția faptului că fereastra document este împărțită în două la jumătate, pentru ca programul să poată afișa mai ușor seturi unice de date în cadrul unei singure vederi. Figura 1256.3 prezintă fereastra Windows *Explorer* și interfața document de tip *Explorer*.

În cadrul programului dumneavoastră veți manipula multe ferestre copil ale unei singure ferestre părinte. Pe măsură ce veți lucra cu ferestre în următoarele secțiuni, veți învăța mai mult despre diferențele importante dintre diversele stiluri de interfață.



***Figura 1256.3** Windows Explorer și interfața document de tip Explorer.*

CREAREA UNUI PROGRAM GENERIC ÎN WINDOWS

C/C++1257

Veți descoperi că majoritatea programelor din Windows execută un anumit volum de prelucrare implicită necesară pentru rularea programului. Mai simplu spus, programul trebuie să creeze cel puțin o fereastră, să înregistreze aceea fereastră în sistemul de operare, iar programul trebuie să gestioneze mesajele trimise de sistemul de operare către fereastră. În plus, majoritatea programelor Windows utilizează un fișier suplimentar denumit *fișier resursă*. Fișierul resursă spune compilatorului caracteristicile ferestrelor produse de program. Secțiunea 1258 explică fișierele resursă în detaliu. După ce ați creat forma de bază a programului Windows, celelalte programe ale dumneavoastră vor folosi această formă de bază ca punct de plecare în crearea programului. Pentru a reduce posibilele confuzii cu următoarele programe, programul de bază îl vom denumi *generic.cpp*. CD-ROM-ul atașat cărții conține toate fișierele componente (dependințele) pentru *generic.cpp*. Totuși, deoarece secțiunile următoare vor explica în detaliu toate dependințele, această secțiune va descrie doar *generic.cpp*:

```
#include <windows.h>
#include "generic.h"
#if defined (win32)

HINSTANCE hInst; // instanta curenta
LPCTSTR lpszAppName = "Generic";
LPCTSTR lpszTitle = "Generic Application";
BOOL RegisterWin95(CONST WNDCLASS* lpwc);

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE
  hPrevInstance,  LPSTR lpCmdLine, int nCmdShow)
  {
    MSG msg;
    HWND hWnd;
    WNDCLASS wc;

    wc.style = CS_HREDRAW | CS_VREDRAW;
```

```
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = 0;
    wc.hIcon = LoadIcon(hInstance, lpszAppName);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = lpszAppName;
    wc.lpszClassName = lpszAppName;

    if(!RegisterWin95(&wc))
     return false;
    hInst = hInstance;
    hWnd = CreateWindow  (lpszAppName,
                          lpszTitle,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, 0,
                          CW_USEDEFAULT, 0,
                          NULL,
                          NULL,
                          hInstance,
                          NULL );
    if(!hWnd)
     return false;
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    while(GetMessage(&msg, NULL, 0,0))
     {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
     }
    return(msg.wParam);
  }
BOOL RegisterWin95(CONST WNDCLASS* lpwc)
  {
    WNDCLASSEX wcex;

    wcex.style = lpwc->style;
    wcex.lpfnWndProc = lpwc->lpfnWndProc;
    wcex.cbClsExtra = lpwc->cbClsExtra;
    wcex.cbWndExtra = lpwc->cbWndExtra;
    wcex.hInstance = lpwc->hInstance;
    wcex.hIcon = lpwc->hIcon;
    wcex.hCursor = lpwc->hCursor;
    wcex.hbrBackground = lpwc->hbrBackground;
    wcex.lpszMenuName = lpwc->lpszMenuName;
    wcex.lpszClassName = lpwc->lpszClassName;
    wcex.cbSize = sizeof(WNDCLASSEX);
```

```
    wcex.hIconSm = LoadIcon(wcex.hInstance, "SMALL");
    return RegisterClassEx(&wcex);
}
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam,
  LPARAM lParam)
{
    switch(uMsg)
    {
      case WM_COMMAND:
        switch(LOWORD(wParam))
        {
          case IDM_TEST :
            break;
          case IDM_EXIT :
            DestroyWindow(hWnd);
            break;
        }
        break;
      case WM_DESTROY :
        PostQuitMessage(0);
        break;
      default:
        return (DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
    return(0L);
}
```

Așa cum vedeți, programul *generic.cpp* constă în trei funcții: *WinMain* (echivalentul lui *main* pentru Windows), *RegisterWin95* și *WndProc*. Așa cum veți vedea în secțiunea 1254 funcția *WndProc* gestionează mesajele trimise programului de către sistemul de operare. Funcția *RegisterWin95* adaugă informație suplimentară la un obiect de tip *WNDCLASS* cerut de Windows 95, nu însă și de Windows NT. În plus, ea înregistrează noua fereastră în sistemul de operare. Veți afla mai multe despre înregistrarea ferestrelor în secțiunile următoare. După compilarea și executarea programului *generic.cpp*, ecranul va afișa conținutul prezentat în Figura 1257.



Figura 1257 *Ieșirea programului **generic.cpp***

1258 FIȘIERELE RESURSĂ

C/C++

Așa cum ați învățat în secțiunea 1255, chiar și cel mai simplu program în Windows are numeroase componente. Cu toate acestea, multe din aceste componente sunt relativ statice, adică programul dumneavoastră nu le va schimba des. Meniurile, de exemplu, se modifică foarte rar în timpul programului. De asemenea, se vor schimba destul de rar, pictogramele, informațiile din barele de titlu și altele. În plus Windows așteaptă ca programele dumneavoastră să stocheze informații referitoare la programe în sine (numele autorului, numărul reviziei etc.) împreună cu programul ca atare. Windows utilizează informația pentru a oferi utilizatorului date utile despre program. De exemplu, dacă selectați *generic.exe* din Windows Explorer și apăsați tastele ALT + ENTER, Windows va afișa caseta de dialog File Properties (Fișier de proprietăți). În plus față de afișarea informațiilor de bază despre fișiere, puteți executa un clic de mouse pe eticheta *Version* din caseta de dialog File Properties care va arăta ca în Figura 1258, de mai jos:



Figura 1258 *Caseta de dialog File Properties a programului generic.exe*

Așa cum vedeți, Windows stochează informația despre fișier în cadrul atributelor proprietăți. În realitate, autorul programului a atribuit aceste proprietăți programului *generic.exe* în cadrul fișierului resursă *generic.rc* (una din dependențele menționate în secțiunea 1257). Fișierul resursă *generic.rc* este un fișier text care încorporează informații despre multe resurse statice ce vor fi folosite de program. Mai în detaliu, fișierul resursă *generic.rc* conține informații despre resursele folosite de program după executare, inclusiv meniuri, pictograme și altele. Fișierul resursă *generic.rc* conține lista pictogramelor, a definițiilor de meniu și a informațiilor despre proprietățile fișierului cum se arată mai jos:

```
#include "windows.h"
#include "generic.h"
#include "winver.h"

MYAPP ICON DISCARDABLE "GENERIC.ICO"
SMALL ICON DISCARDABLE "SMALL.ICO"

GENERIC MENU DISCARDABLE
BEGIN
    POPUP "&File"
```

```
    BEGIN
      MENUITEM "&Test!", IDM_TEST
      MENUITEM "E&xit", IDM_EXIT
    END
    POPUP "&Help"
    BEGIN
      MENUITEM "&About MyApp...", IDM_ABOUT
    END
END

VERSIONINFO
  FILEVERSION 3,3,0,0
  PRODUCTVERSION 3,3,0,0
  FILEFLAGSMASK 0X3fl
#ifdef _DEBUG
    FILEFLAGS 0xbl
#else
    FILEFLAGS 0xal
#endif
  FILEOS 0X4L
  FILETYPE 0X1L
  FILESUBTYPE 0X0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
      BLOCK "040904B0"
      BEGIN
          VALUE "CompanyName", "GenericCompany\0"
          VALUE "FileDescription", "GenericApplication\0"
          VALUE "FileVersion", "1.0\0"
          VALUE "InternalName", "1.0\0"
          VALUE "LegalCopyright", "Copyright \251 Generic
             Company. 1997\0"
          VALUE "LegalTrademarks", "Generic Trademark.\0"
          VALUE "OriginalFilename", "\0"
          VALUE "ProductName", "Generic Application.\0"
          VALUE "ProductVersion", "1.0\0"
      END
    END
    BLOCK "VarFileInfo"
    BEGIN
      VALUE "Translation", 0x409, 1200
    END
END
```

Fişierul resursă *generic.rc* este un simplu fişier de resurse. Pe măsura dezvoltării programului dumneavoastră, veţi descoperi că aţi proiectat un fişier resursă cu sute chiar mii de intrări. Fişierul resursă *generic.rc* gestionează de fapt numai două tipuri de resurse :

informația din cadrul casetei de dialog File Properties, cum ați văzut mai devreme în această secțiune, și informația pentru producerea meniului programului. Când veți folosi compilatorul de resurse pentru a compila fișierul resursă *generic.rc*, compilatorul de resurse va folosi informația stocată aici pentru a genera meniul și informațiile despre proprietățile programului executabil. În fișierul resursă *generic.rc* meniul constă în două meniuri derulante, File și Help, fiecare cu mai multe elemente.

Este important de înțeles că programul nu trebuie să folosească informația furnizată de fișierul resursă. De exemplu, programul *generic.cpp* conține o bară de meniu pentru că a produs o fereastră cu un pointer către bara de meniu *generic* din fișierul *generic.rc*. Implementarea unui fișier resursă nu este suficientă; trebuie implementate și resursele într-un fel sau altul în program.

Observație: *Mulți începători în programarea C++ sub Windows includ facilitatea drag-and-drop folosită pentru a evita construirea de fișiere resursă. Cu instrumentul drag-and-drop puteți adăuga elemente direct în meniu sau în ferestre fără a mai construi manual fișiere cu informații despre aceste resurse. Cu toate acestea însă, proiectarea cu drag-and-drop este unică pentru fiecare compilator. Pentru că practic nu există portabilitate între compilatoare, fișierul resursă este necesar pentru ca programele din această carte să fie consistente pentru toți cititorii.*

1259 IDENTIFICATORI WINDOWS



Așa cum ați învățat în capitolele despre fișiere și I/O, când programul dumneavoastră lucrează cu fișiere și discuri, ele pot face acest lucru fie la un nivel jos cu rutine BIOS, fie la un nivel înalt cu identificatori fișier. În Windows veți folosi identificatori pentru a menține informații despre fișiere. Veți folosi însă un tip deosebit de identificator, denumit identificator de fereastră, pentru a obține sau a menține informații despre ferestrele din program sau alte obiecte din sistem. Un identificator de fereastră este obligatoriu o valoare de tip *long* menținută într-o variabilă de tip *HWND*. Când invocați o funcție API care reclamă un identificator de fereastră, treceți funcției variabila de tip *HWND*. Windows la rândul lui va testa identificatorul de fereastră din lista proprie de identificatori valizi de ferestre și apoi va trimite un mesaj sau va executa o acțiune în fereastra corespunzătoare. Figura 1259 arată modelul logic al procesului de evaluare a identificatorilor de ferestre din Windows.

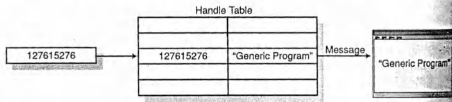


Figura 1259 *Procesul de evaluare a identificatorilor de ferestrele din Windows.*

În cadrul programului dumneavoastră veți folosi identificatori de ferestre pentru a ușura controlul aspectului, dimensiunii sau al altor proprietăți ale unei ferestre.

Așa cum veți învăța în viitoarele secțiuni Windows C++ definește un mare număr de tipuri suplimentare, între care *HWND* este doar unul. De exemplu, programul *generic.cpp* scris în

secțiunea 1257 include șase noi tipuri, pe care programele care nu sunt în Windows nu le pot folosi. Viitoarele secțiuni vor examina aceste noi tipuri în detaliu.

DEFINIREA TIPURILOR DE IDENTIFICATORI WINDOWS

C/C++ 1260

În secțiunea 1259 ați învățat despre tipul *HWND* pe care îl folosiți în programul dumneavoastră Windows pentru a menține o valoare *long* care reprezintă o fereastră deschisă. În fapt, Windows definește 12 tipuri de identificatori. De exemplu, programul *generic.cpp* creat în secțiunea 1257 include o definiție a variabilei de tip *HWND* și a variabilei de tip *HINSTANCE*. Pentru că Windows este un sistem de operare multi-tasking, este posibil să avem mai multe copii sau *instanțe* ale programului, care rulează în același timp. Windows păstrează un singur număr pentru fiecare instanță pe care îl stochează într-o tabelă de identificatori de tip *HINSTANCE*. Programul dumneavoastră poate folosi variabila *HINSTANCE* pentru a menține informația despre instanța care rulează la acel moment. În anumite cazuri, puteți folosi variabila *HINSTANCE* împreună cu un apel Windows API pentru a determina câte instanțe rulează la un moment dat. În secțiunile care au mai rămas în această carte, veți întâlni mai multe tipuri de identificatori. Următoarele secțiuni vor explica în detaliu fiecare tip de identificator pe care îl veți întâlni pe parcursul programului. Tabelul 1260 listează tipurile de identificatori din Windows.

Tip identificator	Descriere
HANDLE	Număr de identificare unic al unui identificator
HBITMAP	Număr de identificare unic al unui bitmap
HBRUSH	Număr de identificare unic al unei pensule
HCURSOR	Număr de identificare unic al unui cursor
HFONT	Număr de identificare unic al unui font
HGDIOBJ	Număr de identificare unic al unui obiect GDI (Graphical Device Interface)
HICON	Număr de identificare unic al unei pictograme
HINSTANCE	Număr de identificare unic al unei instanțe
HPALETTE	Număr de identificare unic al unei palete
HPEN	Număr de identificare unic al unui creion
HRGN	Număr de identificare unic al unei regiuni
HWND	Număr de identificare unic al unei ferestre

Tabelul 1260 *Tipurile de identificatori Windows.*

Este important să înțelegeți că programul dumneavoastră va accesa majoritatea tipurilor de identificatori pentru a controla afișarea informațiilor în cadrul ferestrelor. C++ acceptă suficienți indicatori de fișier pentru Windows, iar noi tipuri de identificatori sunt necesari numai pentru a ușura controlul programului asupra afișării.

1261 FIȘIERUL ANTET GENERIC.H



Înainte de a analiza mai detaliat componentele programului *generic.cpp* este important să observăm în codul originar că programul reclamă utilizarea fișierului *generic.h*:

```
#define IDM_EXIT 100
#define IDM_TEST 200
#define IDM_ABOUT 300

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
```

Fișierul antet *generic.h* definește o serie de constante. Veți recunoaște aceste constante din declarația meniului din fișierul resursă, pentru că numele constantelor corespund cu identificatorii din fișierul resursă. Următoarele secțiuni vor utiliza constantele pentru a stabili când utilizatorul a selectat o opțiune din meniu. Ultimele două linii definesc prototipurile funcțiilor *WndProc* și *About*. Ați realizat funcția *WndProc* în secțiunea 1257. Funcția *About* o veți crea mai târziu. Trebuie să notați, însă, că fișierul antet definește prototipurile celor două funcții ca funcții *callback*. Secțiunea 1262 va prezenta în detaliu funcțiile *callback*.

1262 FUNCȚIILE CALLBACK



În secțiunea precedentă și în secțiunea 1257 ați observat câteva funcții declarate cu cuvântul cheie *CALLBACK*. În continuare veți referi funcțiile pe care programul dumneavoastră le declară cu cuvântul cheie *CALLBACK*, ca *funcții callback*. O funcție callback este o funcție căreia îi transmiteți adresa unei a treia funcții care revine (callback) cu informații. Întotdeauna veți defini funcția *WndProc* ca funcție *callback*. În cadrul programului dumneavoastră veți defini foarte des funcții de tip *callback* împreună cu funcții API specifice, cum sunt *EnumFontFamilies* și *EnumWindows*. Când treceți adresa unei funcții *callback* uneia dintre aceste funcții, funcția respectivă va apela funcția *callback* pentru fiecare element din listă. De exemplu, dacă apelați *EnumWindows*, veți transmite probabil funcției *EnumWindows* adresa unei funcții *callback* care fie va afișa valori, fie le va adăuga unei matrice. *EnumWindows*, în schimb, va apela funcția *callback* pentru fiecare din ferestrele listei cu toate ferestrele.

Funcțiile *callback* sunt necesare în Windows pentru că programul dumneavoastră va gestiona multe dintre acțiunile lor importante prin interfața pentru programarea aplicațiilor (API – Application Program Interface) pe care programul nu o poate modifica direct. Deci trebuie să oferiți interfeței pentru programarea aplicațiilor mijloacele pe care le poate folosi pentru a apela rutinele de utilizator, când returnează o informație extinsă, cum ar fi o listă. Secțiunea 1263 va prezenta în detaliu interfața pentru programarea aplicațiilor a sistemului Windows.

1263 PREZENTAREA INTERFEȚEI PENTRU PROGRAMAREA APLICAȚIILOR SISTEMULUI WINDOWS



Așa cum ați învățat, interfața pentru programarea aplicațiilor a sistemului Windows (API) este fundamentul majorității programelor în Windows. În general, orice acțiune executată de programul dumneavoastră, dincolo de cele mai simple operații matematice, utilizează interfața pentru programarea aplicațiilor fie direct, fie indirect (cu alte cuvinte, chiar definind

un element de meniu într-un fişier resursă se foloseşte Windows API, deşi programul nu apelează funcţiile direct din API). Veţi vedea totuşi, că programul dumneavoastră apelează cel mai adesea direct funcţiile din Windows API, aşa cum se vede mai jos:

```
HWnd = CreateWindow(lpszAppName, lpszTitle, WS_OVERLAPPEDWINDOW,
                    CW_USERDEFAULT, 0, CN_USERDEFAULT, 0, NULL,
                    NULL, hInstance, NULL);
```

Funcţia *CreateWindow* este o funcţie Windows API utilizată de aproape toate programele Windows pentru a realiza ferestrele programului. Aşa cum puteţi vedea, veţi invoca funcţia (şi majoritatea celorlalte funcţii Win32 API) în cadrul programului. Programele dumneavoastră pot apela funcţiile API ca şi cum ar fi definit aceste funcţii, din simplul motiv că toate programele Windows scrise de dumneavoastră vor include fişierul antet *windows.h*. Fişierul antet *windows.h* la rândul lui include multe alte fişiere antet din Windows, în special *winbase.h* care conţine definiţiile funcţiilor Win32 API, ca şi numeroase structuri şi tipuri enumerare (ca *HWND*) folosite în programul dumneavoastră.

Din păcate, Windows API este prea voluminoasă pentru a putea enumera aici funcţiile. În mod curent, sunt peste 1500 de funcţii API de bază şi aproape alte 2000 de funcţii API specifice unui software al sistemului de operare (cum ar fi Microsoft *Internet Explorer 4.0*). Noul sistem de operare Windows 98 are peste 4000 de funcţii API.

DETALII DESPRE PROGRAMUL GENERIC.CPP

C/C++1264

În secţiunea 1257 aţi produs programul *generic.cpp* care implementează cerinţele de bază ale unui program Windows. Înainte de a începe să învăţaţi mai mult despre programarea în Windows şi să începeţi manipularea unor funcţii mai complexe, merită să înţelegeţi exact ce acţiuni execută programul *generic.cpp*. În următoarele câteva secţiuni veţi analiza mai îndeaproape funcţia *WinMain*, procesul de realizare a unei ferestre şi altele. Trebuie, de asemenea, să înţelegeţi semnificaţia variabilelor globale folosite de programul *generic.cpp*, cum se vede mai jos:

```
#include <windows.h>
#include "generic.h"
#if defined (win32)
   #define IS_WIN32 TRUE
#else
   #define IS_WIN32 FALSE
#endif
HINSTANCE hInst; // instanta curenta
LPCTSTR lpszAppName = "Generic";
LPCTSTR lpszTitle = "Generic Application";
BOOL RegisterWin95(CONST WNDCLASS* lpwc);
```

Mai întâi, programul testează constanta compilatorului *win32* pentru a determina dacă acest compilatorul execută sau nu o compilare Win32. Programul verifică dacă este o compilare *Win32* din multe motive, dar în primul rând pentru că, aşa cum aţi învăţat, sunt diferenţe importante între Win32 API şi Win16 API. Un program pe care l-aţi proiectat să ruleze în

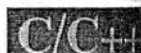
Windows 3.11, dar scris pe o platformă Windows 95, trebuie să se limiteze numai la funcţiile API din Win16 API. Programul dumneavoastră poate folosi constanta *IS_WIN32* pentru a controla ce funcţii vor fi apelate în program.

În al doilea rând, programul defineşte variabila *hInst*. Aşa cum aţi învăţat în secţiunea 1260, *HINSTANCE* este un identificator Windows care menţine un număr unic ce corespunde instanţei programului care se execută la acel moment şi nu o altă instanţă sau alt program.

Se declară apoi *lpszAppName* şi *lpszTitle*, care par la prima vedere a fi matrice de caractere sau variabile de tip şir de caractere. Veţi folosi tipul *LPCTSTR* (un tip definit în Windows) pentru a stoca pointeri *long* protejaţi la scriere (read-only) de tip şir de caractere. Când compilatorul compilează programul, el va converti toate declaraţiile *LPCTSTR* la declaraţii *const char FAR**. Aşa cum vedeţi, însă, *LPCTSTR* este mai uşor de scris şi de înţeles decât *const char FAR**.

În sfârşit, programul face o declaraţie de prototip pentru funcţia *RegisterWin95*. Funcţia *RegisterWin95* acceptă un parametru de tip *WNDCLASS* şi returnează o valoare booleană de reuşită. Veţi învăţa mai mult despre înregistrarea în Windows în secţiunea 1269. Pentru moment, însă, trebuie să înţelegeţi că tipul *WNDCLASS* conţine informaţii pe care Windows le foloseşte de fiecare dată când înregistrează sau produce o nouă fereastră (cum ar fi titlul din bara de titlu, tipul de chenar şi altele).

1265 FUNCŢIA WINMAIN



În programul *generic.cpp* pe care l-aţi scris în secţiunea 1257, prima funcţie a programului a fost *WinMain*. Aşa cum aţi învăţat, funcţia *WinMain* este echivalentul Windows al funcţiei *main* utilizată în toate programele scrise în C şi C++, folosită pentru prelucrări primare. Funcţia *WinMain* diferă însă în multe privinţe faţă de *main* şi nu în ultimul rând, prin modul de declarare, cum este descris mai jos:

```
int APPINTRY WinMain (HINSTANCE hInstance, HINSTANCE
                      hPrevInstance, LPSTR lpCmdLine,
                      int nCmdShow);
```

Aşa cum vedeţi, funcţia *WinMain* returnează o valoare *int* la fel ca multe alte programe în C++. Dar începând de aici similitudinea lor încetează (aşa cum veţi învăţa, antetul funcţiei WinMain execută prelucrări similare cu cele din antetul funcţiei *main*). Cuvântul cheie APIENTRY indică faptul că utilizatorul poate lansa (sau executa) programul din Windows. Tabelul 1265 detaliază parametrii pe care programul dumneavoastră trebuie să-i treacă funcţiei *WinMain*.

Tip parametru	Nume parametru	Descriere
HINSTANCE	*hInstance*	Identificatorul instanţei aplicaţiei. Fiecare instanţă a aplicaţiei deţine un unic identificator. Programatorul va utiliza *hInstance* ca argument al mai multor funcţii Windows şi de asemenea pentru a distinge între multiplele instanţe ale aceleiaşi aplicaţii.

Tip parametru	Nume parametru	Descriere
HINSTANCE	*HPrevInstance*	Instanța precedentă a identificatorului de aplicație. Valoarea lui este *NULL* dacă este prima instanță. În Windows 95 valoarea este întotdeauna *NULL*.
LPSTR	*lpCmdLine*	Un pointer *far* la o linie de comandă terminată în *NULL*. Specificați valoarea *lpCmdLine* când încărcați programul din Program Manager sau când apelați funcția *WinExec*. Rețineți că sub Windows 95 acesta este un pointer la întreaga linie de comandă și nu o matrice de pointeri la fiecare argument (pentru că programul trebuie să analizeze linia de comandă înainte de a începe să o prelucreze).
int	*nCmdShow*	Un întreg care precizează că este afișată fereastra aplicației. Transmiteți această valoare către *ShowWindow*.

Tabelul 1265 *Parametrii acceptați de WinMain.*

În programul *generic.cpp*, prima acțiune a funcției *WinMain* este să definească o variabilă de tip MSG, o variabilă de tip HWND și o variabilă de tip WNDCLASS ca mai jos:

```
MSG msg;
HWND hWnd;
WNDCLASS wc;
```

MSG este un tip enumerare despre care veți afla mai multe în secțiunea următoare. HWND este un identificator de fereastră. Tipul WNDCLASS stochează informația despre clasa Window folosită de program, detaliată în secțiunile 1267 și 1269. Multe din următoarele instrucțiuni din programul *generic.cpp* sunt atribuiri de valori pentru membrii variabilei wc. Secțiunea 1269 va explica în detaliu procesul de atribuire. Următoarele secțiuni vor explica instrucțiunile care au rămas în funcția *WinMain*.

CREAREA FERESTRELOR C/C++1266

Așa cum ați învățat, fundamentul fiecărei interacțiuni a programelor Windows cu utilizatorul se face printr-o fereastră pe care o realizați în program. Fiecare program pe care îl proiectați în Windows pentru a interacționa cu utilizatorul prin interfața Windows va crea cel puțin o fereastră în timpul procesării. Veți învăța mai multe despre înregistrarea claselor fereastră în secțiunea 1269.

Crearea unei ferestre este o operație relativ simplă: mai întâi trebuie să stabiliți componentele și aspectul ferestrei și apoi să utilizați funcția Win32 API *CreateWindow*, pentru a crea efectiv fereastra. Așa cum ați văzut în secțiunea 1257, programul dumneavoastră va implementa funcția *CreateWindow* cu o serie de parametrii. Forma generală a funcției *CreateWindow* este următoarea:

```
HWND CreateWindow(LPCTSTR lpszClassName, LPCTSTR
                  lpszWindowName, DWORD dwStyle, int x, int y,
                  int nWidth, int nHeight, HWND hwndParent,
                  HMENU hmenu, HANDLE hinst LPVOID lpvParam)
```

În mod clar, funcția *CreateWindow* reclamă o anumită elaborare în cadrul programului, înainte ca programul să invoce funcția. Tabelul 1266 detaliază cei 11 parametri ai funcției *CreateWindow*.

Tip parametru	**Nume parametru**	**Descriere**
LPCTSTR	*lpszClassName*	Un pointer constant la un șir terminat în NULL care conține un nume valid de fereastră. Numele clasei poate fi fie cel produs de program cu *RegisterClass*, fie un tip de fereastră predefinită (detalii în secțiunea 1269).
LPCTSTR	*lpszWindowName*	Un pointer constant la un șir terminat în NULL care conține numele ferestrei. În funcție de stilul ferestrei, numele ferestrei poate fi afișat în diverse locații.
DWORD	*dwStyle*	O valoare double-WORD (un întreg de 32 de biți fără semn) care corespunde stilului posibil al ferestrei. Secțiunea 1275 prezintă în detaliu tipul DWORD. Veți crea stiluri din valori combinate în program cu operatorul binar OR. De exemplu: WS_CHILD | ES_LEFT (detalii în secțiunile 1280 și 1280)
int	*x*	Poziția orizontală a colțului din stânga sus al ferestrei. Dacă poziția nu este un element important, parametrul *x* se trece cu valoarea CW_USEDEFAULT.
int	*y*	Poziția verticală a colțului din stânga sus al ferestrei. Dacă poziția nu este un element important, parametrul *y* se trece cu valoarea CW_USEDEFAULT.
int	*nWidth*	Lungimea pe orizontală a ferestrei. Dacă poziția nu este un element important, parametrul *nWidth* se trece cu valoarea CW_USEDEFAULT.
int	*nHeight*	Lungimea pe verticală a ferestrei. Dacă poziția nu este un element important, parametrul *nHeight* se trece cu valoarea CW_USEDEFAULT.
HWND	*hwndParent*	Un identificator al ferestrei părinte. Dacă nu există o fereastră părinte, treceți parametrul cu valoarea NULL.
HMENU	*hmenu*	Un identificator al meniului ferestrei Dacă API folosește meniul înregistrat în clasa ferestrei, treceți parametrul cu valoarea NULL.
HANDLE	*hinst*	Identificatorul instanței programului care a produs fereastra.
LPVOID	*lpvParam*	Un pointer la datele pe care funcția *CreateWindow* le trece în mesajul *WM_CREATE*. Pentru ferestrele copil ale interfeței multiplu-document (MDI) valoarea *lpvparam* trebuie să fie un pointer la o structură *CLIENTCREATESTRUCT*. Pentru majoritatea ferestrelor client non-MDI se trece valoarea NULL.

***Tabelul 1266** Parametrii funcției API **CreateWindow**.*

Dacă crearea unei ferestre vi se pare confuză pe moment, veți descoperi în secțiunea 1267 că multe din ferestrele produse în programe vor partaja caracteristici comune, iar funcția *CreateWindow* va deveni mult mai simplă.

FUNCȚIA CREATEWINDOW

C/C++1267

Secțiunea 1266 v-a prezentat bazele procesului de creare a ferestrelor în programul dumneavoastră. Cu cei 11 parametrii ai săi, totuși, funcția va părea semnificativ mai ușoară dacă luăm în considerare un caz real, în locul cazului general prezentat în secțiunea 1266. Să considerăm, deci, următorul fragment din programul *generic.cpp*:

```
hWnd = CreateWindow  (lpszAppName, lpszTitle, WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL,
                      NULL, hInstance, NULL);
if (!hWnd)
   return false;
```

Cele trei instrucțiuni execută doi pași. Prima instrucțiune încearcă să creeze o fereastră. Dacă instrucțiunea reușește, ea va returna un indicator către noua fereastră produsă. Dacă eșuează, *CreateWindow* va returna *False*. A doua instrucțiune testează identificatorul de fereastră pentru a determina dacă procesul de creare a ferestrei a reușit, iar a treia instrucțiune încheie programul cu rezultatul False, dacă fereastra nu a fost creată.

Înțelegerea procesului de creare a ferestrelor este însă mai complexă. Variabila *lpAppName* indică un șir de caractere care conține numele aplicației, un nume care de asemenea corespunde unor informații conținute în fișierul resursă, precum ați învățat în secțiunile precedente. Pointerul *lpszTitle* corespunde șirului „Generic Application" pe care fereastra îl va afișa în bara de titlu. Parametrul *WS_OVERLAPPEDWINDOW* comunică funcției *CreateWindow* să creeze o fereastră cu suprapunere (*overlapped*). Despre acest stil veți învăța mai mult în secțiunea 1279. Următorii 4 parametrii transmit funcției *CreateWindow* unde să creeze fereastra și cu ce dimensiuni. Primul parametru *NULL* permite funcției *CreateWindow* să știe că fereastra nu are părinte, iar al doilea parametru *NULL* face cunoscut funcției *CreateWindow* că trebuie să încarce meniul implicit al clasei fereastră. Următorii parametri pasează funcției instanța programului, iar ultimul trece valoarea *NULL* mesajului *WM_CREATE*.

Pentru primele dumneavoastră programe, în special, e bine pentru ușurarea prelucrărilor, ca acești parametri să primească valorile implicite sau să folosiți variabile inițializate la începutul programului.

FUNCȚIA SHOWWINDOW

C/C++1268

În secțiunea 1267 ați aflat cum se creează o fereastră în programul *generic.cpp*. Când programați sub Windows însă, după ce creați fereastra trebuie să o faceți vizibilă. Pentru a afișa ferestrele, interfața Win32 API furnizează funcția *ShowWindow* cu următorul prototip:

```
BOOL ShowWindow (HWND hWnd, int nCmdShow);
```

Funcția *ShowWindow* returnează valoarea *true* sau *false* (adevărat sau fals) pe care programul trebuie de regulă să o testeze pentru a determina reușita sau eșecul funcției. Parametrul *hWnd* se referă la fereastra nou creată. Parametrul *nCmdShow* controlează

modul în care programul va afişa fereastra. Valoarea transmisă prin parametrul *nCmdShow* trebuie să corespundă uneia din valorile prezentate în Tabelul 1268.

Valoare	Semnificaţie
SW_HIDE	Ascunde fereastra.
SW_MAXIMIZE	Maximizează fereastra specificată.
SW_MINIMIZE	Minimizează fereastra specificată şi activează fereastra de deasupra din lista de ferestre a sistemului.
SW_RESTORE	Activează şi afişează fereastra. Dacă fereastra curentă este minimizată sau maximizată *ShowWindow* va readuce fereastra în poziţia şi dimensiunea ei iniţială.
SW_SHOW	Afişează fereastra la poziţia şi dimensiunea curentă.
SW_SHOWDEFAULT	Afişează fereastra în starea implicită a aplicaţiei. *ShowWindow* obţine starea implicită a aplicaţiei din structura *STARTUPINFO*, de care veţi afla în secţiunile următoare.
SW_SHOWMAXIMIZED	Afişează fereastra la dimensiunea maximă.
SW_SHOWMINIMIZED	Afişează fereastra la dimensiunea unei pictograme.
SW_SHOWMINNOACTIVE	Afişează fereastra la dimensiunea minimă. Fereastra curentă va rămâne activă.
SW_SHOWNA	Afişează fereastra la dimensiunea curentă. Fereastra curentă va rămâne activă.
SW_SHOWNOACTIVATE	Afişează fereastra la dimensiunea şi poziţia cea mai recentă. Fereastra curentă va rămâne activă
SW_SHOWNORMAL	Afişează fereastra la dimensiunea normală.

Tabelul 1268 *Valori valide pentru parametrii* ***nCmdShow***.

Modificările în programul *generic.cpp* care urmează cer ca Windows să maximizeze fereastra aplicaţiei când utilizatorul selectează opţiunea *Test din* meniul *File*. Ştergeţi codul curent din funcţia *WndProc* şi înlocuiţi-l cu următorul cod:

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam,
  LPARAM lParam)
{
    switch(uMsg)
    {
     case WM_COMMAND:
        switch(LOWORD(wParam))
        {
         case IDM_TEST :
            ShowWindow(hWnd, SW_SHOWMAXIMIZED);
            break;
         case IDM_EXIT :
            DestroyWindow(hWnd);
            break;
        }
        break;
```

```
        case WM_DESTROY :
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
    return(0L);
}
```

Singurul rezultat al schimbării codului inițial din programul *generic.cpp* este că fereastra se maximizează la dimensiunea ecranului atunci când utilizatorul selectează *Test* din meniul ferestrei. După ce compilați și executați programul modificat *generic.cpp* testați prelucrarea programului mai întâi prin maximizarea ferestrei aplicației, apoi aduceți-o la dimensiunea normală. În secțiunile care urmează veți realiza operații mai extinse pentru a controla aspectul ferestrei, utilizând valorile pentru funcția *ShowWindow*.

FUNCȚIA REGISTERCLASS

C/C++1269

Așa cum ați învățat în secțiunile 1266 și 1267, când programul dumneavoastră creează ferestre, ele sunt fie ferestre din clasa predefinită, fie pot dobândi propriul lor stil. Când creați propriul dumneavoastră stil de fereastră, trebuie să înregistrați stilul ferestrei în Windows, înainte de a utiliza stilul pentru a crea ferestrele. Veți utiliza funcția *RegisterClass* din Win32 API pentru a înregistra stilurile de fereastră. Funcția *RegisterClass* are prototipul menționat mai jos:

```
ATOM RegisterClass (CONST WNDCLASS* lpwc);
```

Așa cum puteți vedea, funcția *RegisterClass* returnează o valoare de tip *ATOM*. Tipul *ATOM* este o valoare *WORD* care se referă la un șir de caractere, indiferent dacă sunt cu majuscule sau nu. Faptul că ATOM se referă la șiruri în acest mod înseamnă că „happy“ este echivalent cu „HAPPY“ – echivalență neobișnuită în C++, după cum știți. Windows stochează valori *ATOM* într-o tabelă *ATOM*, iar valoarea *WORD* pe care o menține un *ATOM* este foarte asemănătoare unui identificator.

În plus față de returnarea unei valori *ATOM*, funcția *RegisterClass* acceptă un singur parametru – o constantă pointer la o structură de tip *WNDCLASS*. Windows definește structura *WNDCLASS* după cum urmează:

```
typedef struct tagWNDCLASS
  {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
  } WNDCLASS;
```

Nume Membru	Tip	Funcție
style	*UINT*	Parametrul style trebuie să fie un stil sau o combinație de stiluri realizate cu operatorul binar OR (detalii în Tabelul 1269.2)
lpfnWndProc	*WNDPROC*	Indică o funcție *callback* care prelucrează mesajele pe care Windows le generează pentru ferestre.
cbClsExtra	*int*	Numărul de octeți suplimentari pe care *RegisterClass* trebuie să îi aloce la sfârșitul structurii clasei fereastră pentru stocarea de informații.
cbWndExtra	*int*	Numărul de octeți suplimentari pe care *RegisterClass* trebuie să îi aloce la fiecare creare de instanță pentru stocarea informației.
hInstance	*HINSTANCE*	Un identificator pentru instanța de care aparține clasa fereastra.
hIcon	*HICON*	Un identificator pentru pictograma pe care *CreateWindow* o va folosi pentru această clasă fereastră.
hCursor	*HCURSOR*	Un identificator pentru cursorul pe care *CreateWindow* îl va folosi pentru această clasă fereastră.
hBrush	*HBRUSH*	Un identificator pentru pensula pe care *CreateWindow* o va folosi pentru crearea fundalului. Alte detalii în secțiunile despre pensule.
lpszMenuName	*LPCTSTR*	Un pointer la o constantă șir de caractere terminată în *NULL*, cu numele meniului implicit al clasei. Trebuie să stabiți această valoare ca *NULL* dacă fereastra va avea meniul implicit al clasei.
lpszClassName	*LPCTSTR*	Un pointer la o constantă șir de caractere terminată în *NULL* cu numele clasei. Programul va folosi numele clasei în parametrul lpszClassName al funcției *CreateWindow*.

Tabelul 1269.1 *Datele membre ale structurii WNDCLASS*

Așa cum indică tabelul de mai sus, membrul *style* al clasei fereastră poate corespunde uneia sau mai multor constante binare. Tabelul 1269.2 listează valorile acceptate ale membrului *style*.

Constanta stilului	Semnificație
CS_BYTEALIGNCLIENT	Aliniază zona client a ferestrei pe orizontală în poziția dată de BYTE pentru îmbunătățirea performanței în timpul operațiunilor de desenare. Acest stil afectează lățimea ferestrei și poziția ei orizontală pe ecran.
CS_BYTEALIGNWINDOW	Aliniază fereastra pe orizontală în poziția dată de BYTE.
CS_CLASSDC	Alocă un context de dispozitiv (DC) pentru a fi partajat de toate ferestrele din clasă. Dacă mai multe fire de execuție încearcă să acceseze simultan contextul de dispozitiv, Windows permite doar unui fir să se încheie cu succes. Veți învăța mai târziu despre contextele de dispozitiv.

Constanta stilului	Semnificație
CS_DBLCLKS	Înștiințează o fereastră când utilizatorul execută un dublu clic cu mouse-ul.
CS_GLOBALCLASS	Creează o clasă care este disponibilă pentru toate aplicațiile, cât timp aplicația care a produs clasa este deschisă. În general, se va utiliza când se produc controale personalizate pentru alte programe.
CS_HREDRAW	Redesenează întreaga fereastră dacă utilizatorul ajustează dimensiunea orizontală.
CS_NOCLOSE	Dezactivează comanda Close din meniul de sistem.
CS_OWNDC	Alocă un context de dispozitiv unic pentru fiecare instanță din clasa fereastră.
CS_PARENTDC	Fiecare fereastră produsă în program va utiliza contextul de dispozitiv al ferestrei părinte.
CS_SAVEBITS	Salvează sub forma unui bitmap, porțiunea imaginii din ecran obscurizată de fereastră. Windows va utiliza acest bitmap pentru a reproduce imaginea pe ecran când utilizatorul închide fereastra.
CS_VREDRAW	Redesenează întreaga fereastră dacă utilizatorul ajustează dimensiunea verticală.

Tabelul 1269.2 *Valorile valide pentru membrul* ***style*** *al clasei fereastră.*

Acum, pentru că înțelegeți mai bine cum funcționează instrucțiunea *RegisterClass*, analizați următoarele atribuiri din programul *generic.cpp*, care inițializează clasa fereastră particulară folosită de programul *generic.cpp*:

```
wc.style            = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc      = (WNDPROC)WndProc;
wc.cbClsExtra       = 0;
wc.cbWndExtra       = 0;
wc.hInstance        = hInst;
wc.hIcon            = LoadIcon(hInstance, lpszAppName);
wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName     = lpszAppName;
wc.lpszClassName    = lpszAppName;
```

Prima atribuire comunică sistemului de operare să retraseze întreaga fereastră de fiecare dată când utilizatorul redimensionează fereastra în oricare direcție. A doua atribuire comunică sistemului de operare că funcția *callback* este funcția *WinProc*. Următoarele două atribuiri comunică funcției *RegisterClass* să nu aloce spațiu suplimentar, iar instrucțiunea *wc.hInstance=* transmite compilatorului să utilizeze instanța curentă a programului.

Următoarele două instrucțiuni de atribuire (pentru *hIcon* și *hCursor*) încarcă pictograma și cursorul pentru fereastră. Instrucțiunea de atribuire care urmează, *hbrBackground=*, creează un identificator pentru o pensulă de culoare, iar ultimele două instrucțiuni atribuie numele meniului implicit al ferestrei numele clasei fereastră.

Din nou, pe măsură ce veți învăța majoritatea comenzilor și structurilor pe care le veți manipula în Windows, veți utiliza frecvent comenzile și structurile în același mod, de cele mai multe ori cu aceleași valori și numai ocazional veți modifica unele valori când veți crea o fereastră specială.

***Observație**: Un caz special apare atunci când utilizați **CreateWindow** pentru crearea unei ferestre cu ajutorul unei ferestrei existente. Numele de clase existente sunt **BUTTON, LISTBOX, COMBOBOX, STATIC, EDIT, MDICLIENT** și **SCROLLBAR.** Nu este necesar să înregistrați aceste clase înainte ca programele dumneavoastră să creeze o fereastră utilizând una din aceste clase.*

1270 MAI MULTE DESPRE MESAJE



Ultima funcție din WinMain a programului *generic.cpp* este bucla *while* care prelucrează mesajele din sistem. Așa cum ați învățat, veți scrie programul Windows ca să persiste, de regulă, până când utilizatorul cere ferestrei să se închidă. Orice program Windows pe care îl veți scrie utilizează o *buclă de mesaje (message loop)* pentru continuarea procesării mesajelor până când utilizatorul cere programului să se oprească. Forma standard a unei bucle de mesaje este prezentată mai jos:

```
while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
```

În secțiunile 1271 și 1272 veți învăța mai mult despre *TranslateMessage* și *DispatchMessage*. Este important să înțelegeți mai întâi funcția *GetMessage* și valoarea returnată de ea. Veți folosi funcția *GetMessage* în cadrul programului în prototipul prezentat mai jos:

```
BOOL GetMessage (LPMSG lpmsg, HWND hWnd, UINT uMsgFilterMin,
                 UINT uMsgFilterMax);
```

Funcția *GetMessage* returnează o valoare *true* sau *false* (adevărat sau fals). *GetMessage* returnează *true* până la primirea mesajului *WM_QUIT*. Tabelul 1270 listează parametrii funcției *GetMessage*.

Nume Membru	Tip	Funcție
lpmsg	*MSG*	Returnează un pointer la o structură MSG, prezentată în continuare după acest tabel.
hWnd	*HWND*	Un identificator pentru fereastra care a primit mesajul. De obicei veți stabili această valoare ca *NULL*, ceea ce solicită ca *GetMessage* să rețină toate mesajele pentru firul de execuție curent.
uMsgFilterMin	*UINT*	Valoarea minimă a mesajului primit. De regulă, această valoare se stabilește la 0.
uMsgFilterMax	*UINT*	Valoarea maximă a mesajului primit. Dacă stabiliți ambele valori *uMsgFilterMin* și *uMsgFilterMax* la 0, funcția *GetMessage* va primi toate mesajele.

***Tabelul 1270** Parametrii funcției **GetMessage**.*

Așa cum arată Tabelul 1270, funcția *GetMessage* primește un parametru de tip *MSG*. Windows definește tipul MSG în fișierul antet *winuser.h* prezentat mai jos:

```
typedef struct tagMSG {
    HWND    hwnd;     //identificator pentru fereastra
    UINT    message;  //ID mesaj
    WPARAM  wParam;   //valoare wParam
    LPARAM  lParam;   //valoare lParam
    DWORD   time;     //milisec. dupa pornire
    POINT pt;         //coordonatele locatiei mouse-ului pe ecran
} MSG;
```

Deși fiecare component al structurii *MSG* este important, cel mai frecvent veți manipula membrul *message* care corespunde uneia din numeroasele definiții de constante din Windows pentru mesaje. Veți învăța mai mult despre constantele mesaj în următoarele secțiuni.

UTILIZAREA FUNCȚIEI TRANSLATEMESSAGE PENTRU PRELUCRAREA MESAJELOR

C/C++1271

Așa cum ați învățat în secțiunea 1270, funcția *WinMain* din programul dumneavoastră se termină de obicei cu o buclă *while* care preia mesaje până când utilizatorul trimite sistemului mesajul *WM_QUIT*. În cadrul buclei de mesaje din secțiunea 1270, programul apelează mai întâi funcția *TranslateMessage*. Funcția *TranslateMessage* preia un mesaj de tastă virtuală (cum ar fi VK_TAB) pe care sistemul o generează când utilizatorul apasă o tastă și trimite codul corespunzător *WM_CHAR* în coada de mesaje a aplicației (*WM_CHAR* - abreviere de la *Windows Message Character*). Dacă mesajul nu provine de la o tastă virtuală, *WM_CHAR* va returna valoarea false și nu va prelucra mesajul. Veți utiliza funcția *TranslateMessage* cu prototipul de mai jos:

```
BOOL TranslateMessage (CONST MSG* lpMsg);
```

După cum ați văzut, veți apela funcția *TranslateMessage* imediat după apelul lui *GetMessage*, deși puteți de asemenea apela un mesaj returnat de *PeekMessage* (explicat mai târziu):

```
while (GetMessage (&msg, NULL, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

UTILIZAREA FUNCȚIEI DISPATCHMESSAGE PENTRU PRELUCRAREA MESAJELOR

C/C++1272

În mod normal, în program funcția *WinMain* descrie o buclă, în așteptarea mesajelor. Când un mesaj a sosit, *WinMain* îl expediază unei alte funcții care prelucrează mesaje. Așa cum ați învățat în secțiunea 1270, programul dumneavoastră va crea o buclă de mesaje pentru

prelucrarea mesajelor până când programul se încheie. Ultima componentă a buclei de mesaje este funcția *DispatchMessage* care trimite mesajul funcției de prelucrare după ce instrucțiunea *TranslateMessage* verifică dacă mesajul este de tip Windows. Funcția de prelucrare apelată de *DispatchMessage* este funcția *callback* definită de clasa fereastră în procedura de înregistrare. Implementarea funcției *DispatchMessage* este în general cea descrisă mai jos:

```
long DispatchMessage(CONST MSG* lpMsg);
```

Deși *DispatchMessage* returnează o valoare de tip *long*, programul de regulă ignoră rezultatul pentru că nu aduce informații semnificative programului dumneavoastră.

Observație: *Buclele de mesaje* ***trebuie*** *să includă funcția* ***DispatchMessage***, *altfel nu vor fi capabile să prelucreze mesajele trimise de sistem.*

1273 COMPONENTELE UNUI PROGRAM WINDOWS SIMPLU



În ultimele 16 secțiuni ați studiat programul *generic.cpp* și componentele sale. Programul *generic.cpp* include toate componentele de bază ale oricărui program Windows. Când proiectați programe Windows, trebuie să vă asigurați că programul include toate componentele care urmează, la fel ca și în programul *generic.cpp:*

- Un *fișier resursă.* Deși puteți scrie programe Windows fără folosirea fișierelor resursă, aceasta implică muncă suplimentară pentru dumneavoastră și nu se încadrează în standardul de proiectare a programelor Windows. Toate programele Windows scrise de dumneavoastră trebuie să includă un fișier resursă. Nu trebuie, totuși, să creați mai mult de un fișier resursă pentru un program scris. Rețineți că fișierul resursă poate include informații despre numeroase ferestre și despre conținutul lor în cadrul unui program dat, așa încât trebuie să plasați toate informațiile pentru fiecare program scris într-un singur fișier.
- Un *antet windows.h*. Toate programele Windows trebuie să includă fișierul antet *windows.h*, care cuprinde toate fișierele antet ale tuturor tipurilor, funcțiilor și claselor Windows.
- Funcția *WinMain.* Așa cum toate programele dumneavoastră DOS aveau nevoie de funcția *main*, toate programele Windows trebuie să includă funcția *WinMain.* Rețineți însă că nu puteți utiliza funcția *WinMain* fără cei patru parametri pe care îi așteaptă: HINSTANCE hInstance, *HINSTANCE hPrevInstance, LPSTR lpCmdLine* și *int nCmdShow.*
- O *buclă de mesaje* Windows. Toate programele Windows proiectate pentru interacțiune cu utilizatorul (pe scurt, aproape toate programele) prelucrează mesajele într-o buclă de mesaje. Bucla de mesaje primește mesaje de la coada de mesaje a sistemului și le prelucrează în cadrul funcțiilor programului.
- O funcție *callback de mesaje.* Când creați o fereastră, unul din parametrii pentru acea fereastră indică locul în care ar trebui să-și trimită mesajele în program – funcția callback de mesaje. De asemenea, bucla de mesaje trimite mesaje către funcțiile *callback.* Fiecare program Windows trebuie să aibe o funcție *callback.* În general, veți numi într-un mod consistent funcțiile *callback* de prelucrare a mesajelor, la fel ca și

WinMain, deşi nu reprezintă o cerinţă obligatorie. Veţi avea, de asemenea, funcţii *calback* multiple de prelucrare a mesajelor pentru a gestiona diferit mesajele din diferite programe Windows.

Aproape fiecare program Windows va avea aceste cinci componente. În această carte toate programele Windows prezentate vor avea aceste cinci componente. Dacă proiectaţi un program fără toate aceste cinci componente, veţi avea probabil dificultăţi în funcţionarea corectă a programului sub Windows.

TIPUL LPCTSTR

C/C++1274

Aşa cum aţi învăţat, programul dumneavoastră va folosi tipul Windows LPCTSTR pentru a stoca un pointer de 32 de biţi la o constantă şir de caractere. Tipul de caracter LPCTSTR este portabil atât pentru *Unicode*, cât şi pentru setul de caractere dublu-octet (DBCS – Double-Byte Character Set). Cel mai adesea, şirul LPCTSTR se declară atunci când şirul este un parametru la o funcţie, pe care funcţia nu îl va modifica. Pentru rapiditate şi accesibilitate, când cunoaşteţi că funcţia primeşte un parametru de tip şir de caractere într-un apel *prin valoare*, e bine să declaraţi parametrul şir al funcţiei cu tipul LPCTSTR. În schimb, dacă funcţia returnează valori în cadrul parametrului, ar trebui utilizat parametrul LPTSTR (un pointer de 32 de biţi la un şir de caractere terminat în *NULL*) şi nu LPCTSTR.

De exemplu, următorul fragment de cod declară o clasă *CName*. Clasa *CName* include două funcţii membre *SetData* şi *GetData*. Deoarece funcţia *SetData* nu schimbă informaţii cu şirurile ei componente, ea declară cele două şiruri ca LPCTSTR. În schimb, *GetData* modifică şirurile şi de aceea le declară LPSTR:

```
Class CName
{
private:
   LPSTR m_nPrenume;
   char m_initala;
   CString m_Nume;
public:
   CName() {}
   void SetData(LPCTSTR pr, const char in, LPCTSTR nm)
   {
     m_nPrenume = pr;
     m_initiala = in;
     m_Nume = nm;
   }
   void GetData( LPSTR& spr, char in, LPSTR& snm )
   {
     spr = m_nPrenume;
     in = m_initiala;
     snm = m_Nume;
   }
};
```

1275 TIPUL DWORD

C/C++

Așa cum ați învățat în secțiunile precedente, *DWORD* este un tip dublu-*WORD*. Un tip *WORD* este un întreg fără semn de 16 biți (echivalentul în C pe 16 biți a lui *unsigned long int*). Deci un dublu-*WORD* este un întreg fără semn pe 32 de biți capabil să stocheze valori până la 2^{32}-1. Figura 1275.1 arată modul de alocare a memoriei pe calculator pentru *WORD* și *DWORD*.

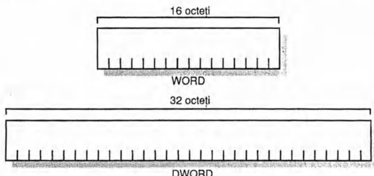


Figura 1275.1 *Modul de alocare a memoriei pentru* ***WORD*** *și* ***DWORD***.

Programul dumneavoastră poate nu numai să folosească *DWORD* pentru numere întregi mari, fără semn, dar cel mai adesea folosește *DWORD* pentru a păstra adrese de segment și de deplasament pe 32 de biți. Deoarece două variabile *WORD* fac un *DWORD*, majoritatea compilatoarelor de C++ oferă numeroase instrumente pentru a separa un *DWORD* într-un *high WORD* (care conține cei doi octeți mai semnificativi) și un *low WORD* (care conține cei doi octeți mai puțin semnificativi). Când separăm un *DWORD* în două variabile *WORD*, *high WORD* reprezintă adresa de segment, iar *low WORD* reprezintă adresa de *deplasament*, ca în figura 1275.2.

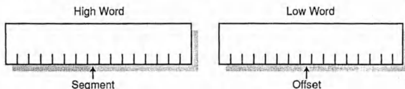


Figura 1275.2 *Un DWORD poate reprezenta o adresă de segment și o adresă de deplasament.*

1276 CLASELE PREDEFINITE DIN WINDOWS

C/C++

Așa cum ați învățat pe scurt în secțiunile 1267 și 1269, puteți crea ferestre de mai multe tipuri prin derivare în cadrul programului dumneavoastră, folosind clasele predefinite în Windows. Ferestrele pe care le veți realiza folosind clasele predefinite, cu excepția ferestrei MDICLIENT (de care veți afla în secțiunile următoare) sunt cunoscute sub numele de *controale*. Alte controale folosite pentru definirea de noi tipuri de ferestre, dar care nu fac obiectul acestei

cărţi, sunt controalele ActiveX şi clasele fereastră generate de compilator. Interfaţa Windows API conţine controalele prezentate în Tabelul 1276.

Clasa	Descriere
BUTTON	Programul dumneavoastră va folosi clasa BUTTON pentru a crea butoane în cadrul ferestrelor. Butoanele pot fi butoane de apăsare în formă de dreptunghi, casete de grup, casete de validare, butoane radio sau pictograme. În general, programul dumneavoastră va folosi butoanele pentru lansarea evenimentelor, pentru comunicarea faptului că s-a terminat completarea unei intrări într-un formular afişat în fereastră etc. Figura 1276.1 afişează un buton cu apăsare într-o fereastră.

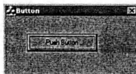


Figura 1276.1 *Un exemplu de buton de apăsare într-o fereastră.*

LISTBOX	Programul dumneavoastră va folosi controlul LISTBOX, pentru a păstra liste cu informaţii pentru utilizatori. Controlul LISTBOX diferă de controlul COMBOBOX pentru că nu acceptă alte intrări decât selecţiile conţinute în lista LISTBOX. Figura 1276.2 arată un exemplu de control LISTBOX.

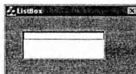


Figura 1276.2 *Un exemplu de control* ***LISTBOX.***

COMBOBOX	Controlul COMBOBOX este un control combinat între un control EDIT şi un control LISTBOX. Veţi folosi controalele COMBOBOX atât pentru a permite utilizatorului să selecteze dintr-o listă, cât şi să introducă propriile date în listă. Figura 1276.3 arată un exemplu de control COMBOBOX

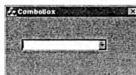


Figura 1276.3 *Un exemplu de control* ***COMBOBOX.***

Clasa	Descriere
STATIC	Programul dumneavoastră va folosi frecvent controlul STATIC, cunoscut și sub denumirea de etichetă, pentru a plasa informații în zona client a ferestrei. Informația plasată cu un control STATIC nu este destinată editării de către utilizator sau modificată în cursul execuției programului. Figura 1276.4 arată un exemplu de control STATIC.

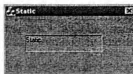


Figura 1276.4 *Un exemplu de control* ***STATIC.***

EDIT	Programul dumneavoastră va folosi controlul EDIT pentru a permite utilizatorilor să introducă informații în program prin interfața Windows. În general, majoritatea programelor Windows vor include unul sau mai multe controale EDIT, în special în programele de aplicații economice. Figura 1276.5 arată un exemplu de control EDIT.

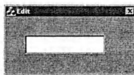


Figura 1276.5 *Un exemplu de control* ***EDIT.***

SCROLLBAR	Programul dumneavoastră va folosi controlul SCROLLBAR pentru a plasa barele de defilare la extremitățile ferestrei, în cadrul zonei client și pentru a interacționa cu utilizatorul.

Tabelul 1276 *Clasele fereastră predefinite.*

În secțiunea 1277 veți realiza un program simplu care folosește mai multe clase fereastră predefinite. În general, veți utiliza frecvent clase fereastră predefinite în cadrul unui singur program și chiar într-o singură fereastră.

Observație: *Atât Microsoft Visual C++, cât și Borland C++ 5.02, cele mai utilizate compilatoare de C++ în Windows, includ suportul de control* ***drag and drop*** *(deplasează și plasează) care vă permite să adăugați controale fără a parcurge pașii suplimentari necesari pentru a realiza controale prin intermediul funcției* ***CreateWindow.***

1277 *UTILIZAREA CLASELOR PREDEFINITE PENTRU A CREA O FEREASTRĂ SIMPLĂ*



În secțiunea 1276 ați învățat despre clasele predefinite pe care le acceptă comanda *CreateWindow*. Așa cum ați învățat, programul dumneavoastră va utiliza în general clase predefinite pentru a crea ferestre copil (controale) într-o fereastră a programului. Folosirea claselor predefinite nu este mai dificilă decât crearea unei instanțe pentru o fereastră deja

înregistrată de program. Programul dumneavoastră va invoca simplu funcția *CreateWindow* în care primul parametru conține numele clasei predefinite, al doilea parametru conține textul din controlul predefinit, iar ultimii parametrii conțin informații despre stilul și poziționarea în fereastra programului.

De exemplu, discul CD-ROM care însoțește cartea de față include un program *show_three.cpp* care creează o fereastră generică pe care ați utilizat-o anterior, la începutul programului. Când utilizatorul selectează opțiunea *Test*, programul afișează un control STATIC, unul *EDIT* și unul *BUTTON*. Programul *show_three.cpp* folosește trei fișiere *show_three.h*, *show_three.cpp* și *show_three.rc*. În general, fișierele sunt aceleași ca pentru *generic.cpp*, cu excepția funcției *WndProc* care apelează *CreateWindow* de trei ori, când utilizatorul selectează opțiunea *Test*.

Așa cum vedeți din listarea codului, singura schimbare între funcția *WndProc* din fișierul *show_three.cpp* și *WndProc* din *generic.cpp* este modul în care programul *show_three.cpp* gestionează selectarea opțiunii *Test* a utilizatorului. În cadrul codului sursă, dacă utilizatorul selectează *Test*, care trimite mesajul *IDM_TEST* funcției *WndProc*, codul va invoca funcția *CreateWindow* de trei ori, o dată pentru controlul *STATIC*, o dată pentru controlul *EDIT* și o dată pentru controlul *BUTTON*.

WINDOWS TRIMITE UN MESAJ ***WM_CREATE*** *CÂND CREEAZĂ O FEREASTRĂ*

C/C++1278

Așa cum ați învățat, de fiecare dată când Windows execută o acțiune semnificativă, el expediază un mesaj către programul care rulează curent, informând programul despre activitatea pe care a încheiat-o. Crearea ferestrelor nu face excepție. De fiecare dată când un program invocă funcția *CreateWindow*, Windows trimite mesaje către funcția *WndProc*. Windows trimite cinci mesaje funcției *WndProc*. De regulă, programul va prelucra mesajul *WM_CREATE* în cadrul funcției *WndProc* și va permite celorlalte patru mesaje să fie transmise către funcția *DefWindowProc*. Aplicația va folosi mesajul *WM_CREATE* pentru a informa programul să inițializeze fereastra. Tabelul 1278 listează cele cinci mesaje trimise de Windows programului când reușește să creeze o fereastră.

Mesaj	**Semnificație**
WM_GETMINMAXINFO	Obține dimensiunea și poziția ferestrei create de Windows.
WM_NCCREATE	Indică faptul că Windows este pe punctul de a crea o zonă ne-client a ferestrei. Funcția *DefWindowProc* alocă memorie pentru fereastră și inițializează barele de defilare când primește mesajul *WM_NCCLIENT*.
WM_NCCALCSIZE	Funcția *DefWindowProc* calculează dimensiunea și poziția zonei client a ferestrei când primește mesajul *WM_NCCCALCSIZE*.
WM_CREATE	Când primește mesajul *WM_CREATE*, care indică faptul că Windows este pe cale de a crea o fereastră, funcția *WndProc* trebuie să execute inițializarea ferestrei.
WM_SHOWWINDOW	Informează funcția *DefWindowProc* că Windows este pe cale de a afișa fereastra.

***Tabelul 1278** Mesaje generate de Windows atunci când programul invocă funcția **CreateWindow**.*

Ca regulă generală, programul trebuie să permită funcției *DefWindowProc* să gestioneze majoritatea proceselor, când creați ferestre. (De fapt, programul trebuie să permită funcției *DefWindowProc* să gestioneze toate mesajele despre care nu specificați să fie gestionate de program.) Așa cum ați văzut, pentru instrucțiunea *case* trebuie să faceți implicit apelul la funcția *DefWindowProc* în funcția *WndProc*, cum se arată mai jos:

```
default:
   return(DefWindowProc(hWnd, uMsg, wParam, lParam));
```

1279 STILURILE FERESTRELOR ȘI CONTROALELOR

C/C++

Așa cum ați învățat în secțiunile precedente, programul dumneavoastră va invoca funcția *CreateWindow* cu un grup de parametri care oferă funcției informații despre noua fereastră de creat. Unul dintre parametrii transmiși funcției *CreateWindow* de fiecare dată când este invocată este parametrul *DWORD dwStyle*. Așa cum ați învățat, programul inițializează valoarea parametrului *dwStyle* folosind o secvență de operatori SAU pe bit care combină o serie de constante cu stilurile ferestrei. Cum ați văzut pe scurt în secțiunea 1277, programul dumneavoastră va folosi parametri *dwStyle* diferiți când creează instanțe ale claselor fereastră predefinite. Tabelul 1279.1 listează parametrii *dwStyle* ai claselor fereastră.

Stil	Descriere
WS_BORDER	Creează o fereastră cu un chenar subțire.
WS_CAPTION	Creează o ferestră cu bară de titlu (include stilul *WS_BORDER*).
WS_CHILD	Creează o fereastră copil. Acest stil nu poate fi folosit împreună cu *WS_POPUP*.
WS_CHILDWINDOW	Creează aceeași fereastră ca și *WS_CHILD*.
WS_CLIPCHILDREN	Exclude zona ocupată de ferestrele copil când se desenează în fereastra părinte. Folosiți acest stil când creați ferestre părinte.
WS_CLIPSIBLINGS	Decupează ferestrele copil una relativ la cealaltă; adică, în momentul în care o anumită fereastră copil primește mesajul WM_PAINT, stilul WS_CLIPSIBLINGS decupează toate celelalte ferestre copil suprapuse din afara regiunii ferestrei copil care va fi actualizată. Dacă stilul WS_CLIPSIBLINGS nu este specificat, iar ferestrele copil se suprapun, e posibil ca în momentul desenării în zona client a unei ferestre copil, să se deseneze eronat în zona client a unei ferestre copil din vecinătate.
WS_DISABLED	Creează o fereastră care este inițial dezactivată. O fereastră dezactivată nu poate primi intrări de la utilizator.
WS_DLGFRAME	Creează o fereastră cu un chenar folosit de obicei la casetele de dialog. O fereastră cu stilul WS_DLGFRAME nu poate avea bară de titlu.

Stil	Descriere
WS_GROUP	Specifică primul control al unui grup de controale. Grupul constă în acest prim control și în toate controalele care sunt definite după el, până la următorul control cu stilul *WS_GROUP*. De obicei primul control în fiecare grup are stilul *WS_TABSTOP*, în așa fel încât utilizatorul să se poată mișca de la un grup la altul. Utilizatorul poate schimba treptat destinația intrărilor de la tastatură de la un control dintr-un grup, la următorul control din grup, folosind tastele de direcție.
WS_HSCROLL	Creează o fereastră cu o bară de defilare orizontală.
WS_ICONIC	Creează o fereastră inițial minimizată. La fel cu *WS_MINIMIZE*.
WS_MAXIMIZE	Creează o fereastră inițial maximizată.
WS_MAXIMIZEBOX	Creează o fereastră care are un buton MAXIMIZE. Programul nu poate combina stilul *WS_MAXIMIZEBOX* cu stilul *WS_EX_CONTEXTHELP*. Programul mai trebuie să specifice stilul *WS_SYSMENU* când indică stilul *WS_MAXIMIZEBOX*.
WS_MINIMIZE	Creează o fereastră inițial minimizată. La fel cu stilul *WS_ICONIC*.
WS_MINIMIZEBOX	Creează o fereastră care are un buton *Minimize*. Programul dumneavoastră nu poate combina acest stil cu stilul *WS_EX_CONTEXTHELP*. Stilul *WS_SYSMENU* trebuie, de asemenea, specificat în program împreună cu stilul *WS_MINIMIZEBOX*.
WS_OVERLAPPED	Creează o fereastră cu suprapunere. O fereastră cu suprapunere are o bară de titlu și un chenar. La fel cu stilul *WS_TILED*.
WS_OVERLAPPEDWINDOW	Creează o fereastră cu suprapunere care include stilurile *WS_OVERLAPPED*, *WS_CAPTION*, *WS_SYSMENU*, *WS_THICKFRAME*, *WS_MINIMIZEBOX* și *WS_MAXIMIZEBOX*. Similar cu stilul *WS_TILEDWINDOW*.
WS_POPUP	Creează o fereastră pop-up. Nu puteți folosi *WS_POPUP* cu stilul *WS_CHILD*.
WS_POPUPWINDOW	Creează o fereastră pop-up, care include stilurile *WS_BORDER*, *WS_POPUP* și *WS_SYSMENU*. Trebuie combinate stilurile *WS_CAPTION* și *WS_POPUPWINDOW* pentru a face meniul vizibil.
WS_SIZEBOX	Creează o fereastră cu un chenar care permite dimensionarea ferestrei. Similar cu stilul *WS_THICKFRAME*.
WS_SYSMENU	Creează o fereastră care are un meniu în bara de titlu. Stilul *WS_CAPTION* trebuie, de asemenea, specificat.
WS_TABSTOP	Specifică faptul că un control poate primi focusul pentru intrări de la tastatură atunci când utilizatorul apasă pe tasta TAB. Apăsarea tastei TAB schimbă ținta intrărilor de la tastatură la următorul control cu stilul *WS_TABSTOP*.

(continuare)

Stil	Descriere
WS_THICKFRAME	Creează o fereastră cu un chenar care permite dimensionarea ferestrei. Similar cu stilul *WS_SIZEBOX*.
WS_TILED	Creează o fereastră cu suprapunere. O fereastră cu suprapunere are o bară de titlu și un chenar. Similar cu stilul *WS_OVERLAPPED*.
WS_TILEDWINDOW	Creează o fereastră cu suprapunere, care include stilurile *WS_OVERLAPPED*, *WS_CAPTION*, *WS_SYSMENU*, *WS_THICKFRAME*, *WS_MINIMIZEBOX* și *WS_MAXIMIZEBOX*. Similar cu stilul *WS_OVERLAPPEDWINDOW*.
WS_VISIBLE	Creează o fereastră inițial vizibilă.
WS_VSCROLL	Creează o fereastră cu o bară de defilare verticală.

Tabelul 1279.1 *Valorile posibile ale parametrului* ***dwStyle*** *când creați ferestre în program.*

Așa cum ați învățat, programul dumneavoastră poate folosi diferite valori pentru clasele fereastră predefinite din Windows, când creați controale sau alte tipuri de ferestre. Tabelul 1279.2 detaliază valorile posibile ale parametrului dwStyle când creați o fereastră de tip *BUTTON* în cadrul programului.

Stil	Descriere
BS_3STATE	Creează un buton similar unei casete de validare, cu excepția faptului că butonul poate fi dezactivat, validat sau nevalidat. Folosiți starea dezactivată pentru a arăta că starea casetei de validare nu este determinată.
BS_AUTO3STATE	Creează un buton similar cu o casetă de validare cu trei stări, cu următoarea excepție: caseta își schimbă starea când utilizatorul a selectat-o. Starea ciclează între validat, dezactivat și nevalidat.
BS_AUTOCHECKBOX	Creează un buton similar cu o casetă de validare, cu următoarea excepție: starea de validare se comută automat între validat și nevalidat de fiecare dată când utilizatorul selectează caseta de validare.
BS_AUTORADIOBUTTON	Creează un buton similar cu un buton radio, cu excepția faptului că atunci când utilizatorul îl selectează, Windows stabilește automat starea de validare pe validat și stabilește automat starea de validare pe nevalidat la toate celelalte butoane din același grup.
BS_CHECKBOX	Creează o casetă de validare mică, goală, cu text. Implicit, textul este afișat în dreapta casetei de validare. Pentru a afișa textul la stânga casetei de validare, combinați acest stil cu stilul *BS_LEFTTEXT* (sau stilul său echivalent *BS_RIGHTBUTTON*).
BS_DEFPUSHBUTTON	Creează un buton cu apăsare care se comportă ca BS_PUSHBUTTON, dar are un chenar negru îngroșat. Dacă butonul este într-o casetă de dialog, utilizatorul poate selecta butonul prin apăsarea tastei ENTER, chiar când butonul nu este ținta intrărilor. Acest stil este folositor pentru a permite utilizatorului să selecteze rapid cea mai adecvată (implicită) opțiune.

Stil	**Descriere**
BS_GROUPBOX	Creează un dreptunghi în care pot fi grupate alte controale. Toate textele asociate acestui stil sunt afişate în colţul din stânga sus al dreptunghiului. *GROUPBOX* mai este cunoscut de obicei şi ca un cadru (frame).
BS_LEFTTEXT	Plasează text în partea stângă a butonului radio sau casetei de validare când se combină cu stilurile buton radio sau casetă de validare. Similar cu stilul *BS_RIGHTBUTTON*
BS_OWNERDRAW	Creează un buton care poate fi desenat de utilizator. Fereastra deţinătoare primeşte mesajul WM_MEASUREITEM când Windows creează butonul şi nu mesaj *WM_DRAWITEM* când aspectul vizual al butonului s-a modificat. Nu combinaţi acest stil BS_OWNERDRAW cu nici un alt stil de buton.
BS_PUSHBUTTON	Creează un buton de apăsare care expediază mesajul WM_COMMAND ferestrei deţinătoare când utilizatorul a selectat butonul.
BS_RADIOBUTTON	Creează un mic cerc asociat unui text. În mod implicit, textul este afişat în dreapta cercului. Pentru a afişa textul în stânga cercului, combinaţi acest stil cu stilul BS_LEFTTEXT (sau cu echivalentul său BS_RIGHTBUTTON). Butoanele radio se folosesc în grupuri de opţiuni corelate, dar care se exclud reciproc.
BS_USERBUTTON	Stilul este în prezent depăşit, dar este păstrat pentru compatibilitatea cu versiunile Windows pe 16 biţi. Aplicaţiile bazate pe Win32 vor folosi stilul *BS_OWNERDRAW*.
BS_BITMAP	Specifică faptul că butonul afişează un bitmap.
BS_BOTTOM	Plasează text în partea de jos a dreptunghiului butonului.
BS_CENTER	Centrează textul pe orizontală în dreptunghiul butonului.
BS_ICON	Specifică faptul că butonul afişează o pictogramă.
BS_LEFT	Aliniază la stânga textul în dreptunghiul butonului. Dacă însă, butonul este o casetă de validare sau un buton radio care nu are stilul BS_RIGHTBUTTON, textul este aliniat la stânga în partea dreaptă a casetei de validare sau a butonului radio.
BS_MULTILINE	Desfăşoară textul pe mai multe linii, dacă şirul de text este prea lung pentru a încăpea pe o singură linie a dreptunghiului butonului.
BS_NOTIFY	Permite butonului să trimită mesajele de notificare BN_DBLCLK, BN_KILLFOCUS şi BN_SETFOCUS ferestrei părinte a butonului. Reţineţi că butoanele trimit mesajul de notificare BN_CLICKED indiferent dacă are sau nu stilul BS_NOTIFY.
BS_PUSHLIKE	Face ca un buton (cum sunt caseta de validare, caseta de validare cu trei stări sau butonul radio) să arate şi să acţioneze ca un buton cu apăsare. Butonul apare ridicat când nu este apăsat sau validat şi înfundat când este apăsat sau validat.

(continuare)

Stil	Descriere
BS_RIGHT	Aliniază la dreapta textul în dreptunghiul butonului. Dacă însă, butonul este o casetă de validare sau un buton radio care nu au stilul BS_RIGHTBUTTON, textul este aliniat la dreapta în partea dreaptă a casetei de validare sau a butonului radio.
BS_RIGHTBUTTON	Poziționează cercul butonului radio sau pătratul casetei de validare în partea dreaptă a dreptunghiului butonului. Similar cu stilul BS_LEFTTEXT.
BS_TEXT	Specifică faptul că un buton afișează text.
BS_TOP	Plasează text în partea de sus a dreptunghiului butonului.
BS_VCENTER	Plasează text în partea centrală a dreptunghiului butonului.

Tabelul 1279.2 *Valorile posibile ale parametrului **dwStyle** când creați ferestre de tip buton.*

Așa cum fereastra BUTTON are propriile ei stiluri, la fel are fiecare clasă fereastră predefinită Acest capitol nu-și propune să listeze toate stilurile acestora pentru că spațiul necesar ar fi mult prea mare.

1280 CREAREA FERESTRELOR CU STILURI EXTINSE



În secțiunile precedente ați învățat cum să utilizați funcția *CreateWindow* pentru a realiza în programele dumneavoastră atât ferestre predefinite, cât și ferestre definite de utilizator. În mod similar, programul dumneavoastră poate utiliza funcția *CreateWindowExt* pentru a crea ferestre cu *stiluri extinse* cum ar fi ferestre cu suprapunere, pop-up și ferestre copil. În programele dumneavoastră funcția *CreateWindowExt* va fi utilizată astfel:

```
HWND CreateWindowEx{
  DWORD dwExStyle,       // stilul ferestrei extinse
  LPCTSTR lpClassName,   // pointer la numele clasei
                         // inregistrate
  LPCTSTR lpWindowName,  // pointer la numele ferestrei
  DWORD dwStyle,         // stilul ferestrei
  int x,                 // coordonata orizontala a ferestrei
  int y,                 // coordonata verticala a ferestrei
  int nWidth,            // latimea ferestrei
  int nHeight,           // inaltimea ferestrei
  HWND hWndParent,       // identificator pentru fereastra
                         // parinte sau detinatoare
  HMENU hMenu,           // identificator pentru meniu sau
                         // fereastra copil
  HINSTANCE hInstance,   // identificator pentru instanta
                         // aplicatiei pointer catre datele
  LPVOID lpParam         // ferestrei
}
```

Așa cum puteți vedea, funcția *CreateWindowExt* acceptă aceiași parametri cu cei ai funcției *CreateWindow*, cu excepția parametrului *dwExStyle* care specifică stilul extins al ferestrei. Rețineți că programul dumneavoastră poate specifica un număr nelimitat de stiluri *dwExStyle*

pentru ferestre combinate folosind operatorul SAU (OR) pe bit. Tabelul 1280 listează valorile posibile ale stilului *dwExStyle*.

Stil	Descriere
WS_EX_ACCEPTFILES	Specifică faptul că fereastra produsă cu acest stil acceptă tehnica manevrării fişierelor cu facilitatea *drag-and-drop*.
WS_EX_APPWINDOW	Forţează o fereastră pe bara de taskuri când Windows minimizează fereastra.
WS_EX_CLIENTEDGE	Specifică faptul că fereastra are un chenar cu aspect tridimensional.
WS_EX_CONTEXTHELP	Inserează semnul de întrebare în bara de titlu a ferestrei. Când utilizatorul execută clic cu mouse-ul pe semnul de întrebare, cursorul ia aspectul unui semn de întrebare şi o săgeată. Dacă atunci utilizatorul execută clic cu mouse-ul pe o fereastră copil, aceasta primeşte mesajul WM_HELP. Fereastra copil trebuie să treacă mesajul procedurii fereastră a ferestrei părinte, care trebuie să apeleze funcţia *WinHelp* folosind comanda *HELP_WM_HELP*. Aplicaţia help afişează o fereastră pop-up care conţine informaţiile de help ale ferestrei copil. Programul dumneavoastră nu poate folosi WS_EX_CONTEXTHELP cu stilurile WS_MAXIMIZEBOX sau WS_MINIMIZEBOX.
WS_EX_CONTROLPARENT	Permite utilizatorului să navigheze printre ferestrele copil cu tasta TAB.
WS_EX_DLGMODALFRAME	Desemnează o fereastră cu chenar dublu. Opţional poate primi o bară de titlu dacă este specificat stilul WS_CAPTION în parametrul *dwStyle*.
WS_EX_LEFT	Conferă ferestrei stilul generic de aliniere la stânga. Este stilul implicit.
WS_EX_LEFTSCROLLBAR	Plasează bara de defilare verticală (dacă există) în stânga zonei client, dacă limba sistemului este ebraica, araba sau altă limbă care acceptă alinierea ordinii de citire. Pentru alte limbi, Windows ignoră stilul şi nu îl tratează ca pe o eroare.
WS_EX_LTRREADING	Afişează textul în fereastră în ordinea stânga-dreapta. Este ordinea implicită.
WS_EX_MDICHILD	Creează o fereastră copil MDI..
WS_EX_NOPARENTNOTIFY	Specifică faptul că o fereastră copil creată cu acest stil nu va trimite mesajul WM_PARENTNOTIFY către fereastra părinte când fereastra copil este creată sau eliminată.
WS_EX_OVERLAPPEDWINDOW	Combină stilurile WS_EX_CLIENTEDGE şi WS_EX_WINDOWEDGE.
WS_EX_PALETTEWINDOW	Combină stilurile WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW şi WS_EX_TOPMOST.

(continuare)

Stil	Descriere
WS_EX_RIGHT	Windows are proprietăți generice de aliniere la dreapta, deși proprietatea depinde de clasa fereastră. Acest stil funcționează numai dacă limba sistemului este ebraica, araba sau o altă limbă ce acceptă alinierea ordinii de citire; altfel, Windows ignoră stilul și nu îl tratează ca pe o eroare. Utilizând stilul WS_EX_RIGHT pentru controalele Static și Eolit se obține același efect ca și atunci când se utilizează SS_RIGHT sau ES_RIGHT. Utilizând acest stil cu controalele Button se obține același efect ca și atunci când se utilizează BS_RIGHT și BS_RIGHTBUTTON.
WS_EX_RIGHTSCROLLBAR	Plasează bara de defilare verticală (dacă este prezentă) în partea dreaptă a zonei client. Stilul este implicit.
WS_EX_RTLREADING	Afișează textul ferestrei în ordinea de citire dreapta-stânga, dacă limba sistemului este ebraica, araba sau o altă limbă ce acceptă alinierea ordinii de citire; altfel, Windows ignoră stilul și nu îl tratează ca pe o eroare.
WS_EX_STATICEDGE	Produce o fereastră cu un stil de chenar tridimensional, folosit pentru elemente care nu acceptă intrări de la utilizator.
WS_EX_TOOLWINDOW	Creează o fereastră de instrumente cu o fereastră proiectată să fie utilizată ca bară de instrumente flotantă. O fereastră de instrumente are o bară de titlu mai mică decât cea normală, iar titlul ferestrei este scris cu un font mic. Fereastra de instrumente nu apare în bara de taskuri sau în caseta de dialog care apare când utilizatorul apasă pe ALT+TAB. Dacă o fereastră cu instrumente are un meniu de sistem, Windows nu îi afișează pictograma pe bara de tiltu. Totuși, puteți executa clic-dreapta sau apăsa pe ALT+SPAȚIU pentru a afișa meniul sistem.
WS_EX_TOPMOST	Fereastra creată cu acest stil va trebui plasată deasupra tuturor ferestrelor care nu au acest stil și va sta deasupra lor chiar dacă fereastra este dezactivată. O aplicație poate folosi funcția membru *SetWindowPos* pentru a fixa sau elimina acest atribut.
WS_EX_TRANSPARENT	Fereastra creată cu acest stil va fi transparentă. Aceasta înseamnă că toate ferestrele care sunt sub ea vor fi vizibile. Fereastra cu acest stil va primi mesajul *WM_PAINT* numai după ce toate ferestrele similare de dedesubt au fost actualizate.
WS_EX_WINDOWEDGE	Specifică faptul că o fereastră are un chenar în relief.

***Tabelul 1280** Valorile posibile ale parametrului **dwExStyle**.*

Pe măsură ce programul dumneavoastră câștigă în complexitate și inserați din ce în ce mai multe ferestre în el, veți folosi *CreateWindowEx* mai frecvent decât *CreateWindow*. De exemplu, următoarea instrucțiune creează o fereastră cu o bară mică de titlu (cum ar fi o fereastră de instrumente) și nu o fereastră cu bară de titlu normală.

```
hwndMaain = CreateWindowEx(WS_EX_SMCAPTION, lpszaAppname,
                           lpszTitle, WS_OVERLAPPEDWINDOW |
                           WS_CLIPCHILDREN, CW_USEDEFAULT, 0,
                           CW_USEDEFAULT, 0, NULL, NULL,
                           hInstance,NULL );
```

ELIMINAREA FERESTRELOR

C/C++ 1281

În ultimele secțiuni ați învățat cum să creați ferestre definite de utilizator, ferestre predefinite și ferestre cu stiluri extinse. Așa cum știți, orice instanță a unei ferestre este un obiect al unei clase fereastră și, cum ați învățat anterior, programul dumneavoastră trebuie întotdeauna să distrugă obiectele după ce programul a încheiat prelucrarea acelui obiect. Deoarece însă, dumneavoastră nu folosiți *new, malloc* sau o funcție similară pentru a aloca memorie când programul construiește fereastra, nu puteți să folosiți funcția *delete* sau o altă funcție de ștergere a memoriei și distrugere a ferestrei. În schimb, puteți apela funcția API *DestroyWindow*. Funcția *DestroyWindow* șterge fereastra pe care o transmiteți în singurul ei parametru. Următorul prototip arată cum veți folosi funcția *DestroyWindow* în programul dumneavoastră:

```
BOOL DestroyWindow (HWND hWnd);
```

Funcția *DestroyWindow* returnează True dacă reușește sau False în caz contrar. Înainte de a distruge fereastra, *DestroyWindow* trimite mesajele *WM_DESTROY* și *WM_NCDESTROY* către fereastră pentru a o dezactiva. Procedura fereastră va răspunde mesajelor *WM_DESTROY* și *WM_NCDESTROY* înainte ca programul să distrugă fereastra. *DestroyWindow* distruge meniul ferestrei și eliberează firul din coada de mesaje.

În plus față de folosirea funcției *DestroyWindow* pentru a distruge ferestrele construite cu funcția *CreateWindow*, programul dumneavoastră poate de asemenea folosi *DestroyWindow* pentru a distruge casetele de dialog nemodale create cu funcția *CreateDialog*. Totuși, în mod curent veți utiliza funcția *DestroyWindow* în rutina *WndProc* pentru a răspunde unei comenzi de ieșire (Exit) din cadrul programului. Următorul cod arată cum folosesc programele *generic.cpp* și *show_three.cpp* opțiunea de meniu File – Exit pentru a distruge fereastra programului:

```
case IDM_EXIT:
   DestroyWinndow(hWnd);
   break;
```

FUNCȚIA API REGISTERCLASSEX

C/C++ 1282

În secțiunea 1269 ați învățat că programele dumneavoastră folosesc funcția API *RegisterClass* pentru a înregistra o clasă definită de utilizator. Funcția *RegisterClassEx* diferă de *RegisterClass* numai în aceea că vă permite să înregistrați fereastra cu o pictogramă pe care o va plasa în bara de titlu în fiecare instanță a clasei înregistrate. Veți utiliza funcția *RegisterClassEx* cu următorul prototip:

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);
```

Funcția *RegisterClassEx* returnează tipul ATOM ca și *RegisterClass*. Totuși, *RegisterClassEx* primește un singur parametru, un pointer la structura *WNDCLASSEX* (și nu *WNDCLASS* pe care o primește *RegisterClass*). Trebuie să completați structura cu atributele corespunzătoare înainte de a o transmite funcției. Structura *WNDCLASSEX* este puțin diferită de structura *WNDCLASS*, cum se arată mai jos:

```
typedef struct tagWNDCLASSEX
  {
    UINT      style;
    WNDPROC   lpfnWndProc;
    int       cbClsExtra;
    int       cbWndExtra;
    HINSTANCE hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH hbrBackground;
    LPCTSTR   lpszMenuName;
    LPCSTR lpszClassName;
    HICON     hIconSm;
  } WNDCLASSEX;
```

Singura adăugare pe care o face *WNDCLASSEX* la structura *WNDCLASS* este identificatorul *HICOM* ca ultim element. Când inițializați structura *WNDCLASSEX* veți folosi comanda *LoadIcon* pentru a atribui o valoare membrului *hIconSm*.

Observație: *Sub Windows 95 funcția* ***RegisterClass*** *eșuează dacă membrul* ***cbWndExtra*** *sau* ***cbClsExtra*** *al structurii* ***WNDCLASSEX*** *conține mai mult de 40 de octeți.*

1283 ATAȘAREA INFORMAȚIILOR ÎN FEREASTRĂ CU SETPROP

C/C++

Așa cum ați învățat, crearea ferestrelor în programele dumneavoastră este un proces relativ simplu. În plus față de crearea ferestrelor, programul dumneavoastră poate atașa în fereastră o listă de date asociate, cunoscute ca *articole de proprietăți*. În general, veți atașa articole de proprietăți într-o fereastră pentru a facilita accesul programului dumneavoastră la informații fără utilizarea variabilelor globale. Fiecare secțiune din program care recunoaște sau poate accesa identificatorul unei anumite ferestre, poate apoi obține informații pe care le-ați atașat anterior la fereastră cu ajutorul listei de proprietăți. În cadrul programului dumneavoastră veți utiliza funcția *SetProp* pentru a adăuga o nouă intrare în lista cu articole de proprietăți sau a schimba una deja existentă pentru o anumită fereastră. Funcția *SetProp* inserează o nouă intrare în listă dacă respectivul șir de caractere nu există în listă. Noua intrare conține șirul și identificatorul. În caz contrar, funcția înlocuiește identificatorul curent al șirului cu identificatorul specificat. Funcția *SetProp* se va folosi în program ca în exemplul de mai jos:

```
BOOL SetProp(
    HWND hWnd,          //identificator pentru fereastra
    LPCTSTR lpString,   //atom sau adresa a sirului
    HANDLE hDate        // identificator pentru date
);
```

Funcția *SetProp* returnează *True* dacă a reușit și *False*, în caz contrar. Tabelul 1283 listează parametrii funcției *SetProp* și descrierea lor.

Parametru	Descriere
hWnd	Identifică fereastra a cărei listă de proprietăți primește o nouă intrare.
lpString	Indică un șir de caractere terminat cu NULL sau conține un ATOM care identifică un șir. Dacă acest parametru este un ATOM, el trebuie să fie un ATOM global, produs anterior prin apelul la *GlobalAddAtom*. Progra- mul trebuie să treacă parametrului ATOM o valoare de 16 biți în cuvântul (Word) mai puțin semnificativ al parametrului *lpString*. Valoarea cea mai semnificativă trebuie să fie zero.
hData	Identifică datele funcției *SetProp* pentru a le copia în lista de proprietăți. Parametrul *hData* poate identifica orice valoare utilă pentru aplicație.

***Tabelul 1283** Parametrii funcției **SetProp**.*

Înainte de a distruge o fereastră (adică înainte de a prelucra mesajul *WM_DESTROY*) aplicația trebuie să șteargă toate intrările adăugate în lista de proprietăți. Aplicația trebuie să folosească funcția *RemoveProp* pentru a șterge aceste intrări. Programul *show_prop.cpp* inclus în CD-ROM-ul care însoțește cartea de față, stabilește o proprietate a ferestrei la crearea ei, apoi afișează proprietatea când utilizatorul a selectat opțiunea *Test*. Ca și programul anterior, singura diferență dintre fișierul *generic.cpp* și *show_prop.cpp* se află în funcția *WndProc*.

Codul din cadrul funcției *WndProc* pentru programul *show_prop.cpp* execută multe lucruri noi. Mai întâi, funcția *WndProc* adaugă o instrucțiune *case* la mesajul *WM_CERATE*. Când programul primește mesajul *WM_CREATE* el adaugă o proprietate cu șirul *Value for Property* în lista de proprietăți a ferestrei. Când selectați mai târziu opțiunea *Test* prðgramul va afișa numai această singură cunoscută proprietate în caseta de mesaje. Veți învăța despre caseta de mesaje (Message Box) în secțiunea 1286. Secțiunea 1284 detaliază cum puteți lista proprietățile ferestrei când programul nu știe cheia reală pentru proprietate.

UTILIZAREA FUNCȚIEI ENUMPROPS PENTRU LISTAREA PROPRIETĂȚILOR FERESTRELOR

C/C++1284

În secțiunea 1283 ați învățat cum să utilizați funcția *SetProp* pentru a stabili o proprietate a ferestrei și, pe scurt, cum să folosiți funcția *GetProp* pentru a returna acea proprietate în altă parte a programului. Așa cum vedeți, funcția *GetProp* cere ca funcția să transmită fie un pointer la un șir de caractere terminat cu NULL, fie un ATOM care identifică proprietatea pe care vreți să o prelucrați. Sunt situații în care programul trebuie să primească toți parametrii asociați cu o fereastră, fără să cunoaștem numele proprietăților, câte proprietăți sunt etc. Funcția *EnumProp* vă permite să enumerați toate intrările din lista de proprietăți a unei ferestre prin transmiterea fiecărui articol în parte către funcția *callback* respectivă. *EnumProps* continuă până când enumeră ultima intrare sau funcția *callback* returnează *False*. În cadrul programului, veți invoca funcția *EnumProps* în următoarea formă generală:

```
int EnumProps(
    HWND hWnd;              //identificator pentru fereastra
    PROPENUMPROC lpEnumProc; // pointer la functia callback
);
```

Înaintea invocării funcției *EnumProps*, trebuie să definiți funcția *callback* pe care o invocă *EnumProps*. Formatul general al funcției *callback* este arătat mai jos (notați că puteți numi funcția cu ce nume doriți, antetul *PropEnumProc* este doar un înlocuitor):

```
BOOL CALLBACK PropEnumProc(
   HWND hWnd;           //identificator pentru fereastra
   LPCTSTR lpszString;  //sirul proprietate
   HANDLE hData;        //identificator pentru date
);
```

Se impun următoarele restricții funcției *callback PropEnumProp:*

1. Funcția callback nu trebuie să transfere controlul sau să acționeze în așa fel încât să transfere controlul altor taskuri.
2. Funcția callback poate apela funcția *RemoteProp*. Însă, *RemoveProp* poate elibera numai proprietatea transmită funcției callback prin parametrii ei.
3. Funcția callback nu trebuie să încerce să adauge proprietăți.

Când puneți funcțiile *EnumProps* și *PropEnumProc* împreună, programul dumneavoastră poate lista fiecare proprietate pe care anterior ați asociat-o cu o fereastră în aceeași ordine în care programul a asociat proprietățile în fereastră. Compact discul care însoțește cartea de față include programul *EnumProps.cpp* care modifică ușor programul WndProc și adaugă o nouă funcție – *EnumPropsProc* la programul *generic.cpp*. Pentru a realiza cât mai corect funcția *EnumProps* ștergeți funcția *WndProc* existentă din programul generic.cpp și în locul ei adăugați următorul cod:

```
BOOL CALLBACK EnumPropsProc( HWND hWnd, LPCTSTR lpszString,
  HANDLE hData )
  {
   MessageBox( hWnd, (LPCTSTR)hData, lpszString, MB_OK );
   return( TRUE );
  }
LRESULT CALLBACK WndProc(HWND hWnd,UINT uMsg,WPARAM wParam,
           LPARAM lParam )
  {
   static LPCTSTR szProp1 = "Value for Property 1";
   static LPCTSTR szProp2 = "Value for Property 2";
   static LPCTSTR szProp3 = "Value for Property 3";
   switch( uMsg )
     {
      case WM_CREATE :
        // Adauga articole de proprietate care contin
        // pointeri de siruri la fereastra principala
        SetProp( hWnd, "Property 1", (HANDLE)szProp1 );
        SetProp( hWnd, "Property 2", (HANDLE)szProp2 );
        SetProp( hWnd, "Property 3", (HANDLE)szProp3 );
        break;
```

```
      case WM_COMMAND :
        switch( LOWORD( wParam ) )
        {
          case IDM_TEST :
            // enumera proprietatile si afiseaza intr-o
            // caseta de mesaje numele proprietatii si valoarea ei.
            EnumProps( hWnd, (PROPENUMPROC)EnumPropsProc );
            break;
          case IDM_ABOUT :
            DialogBox( hInst, "AboutBox", hWnd, (DLGPROC)About );
            break;

          case IDM_EXIT :
            DestroyWindow( hWnd );
            break;
        }
        break;

      case WM_DESTROY :
        PostQuitMessage(0);
        break;
      default :
        return( DefWindowProc( hWnd, uMsg, wParam, lParam ) );
    }
  return( 0L );
}
```

Când compilați și executați programul *EnumProps* și selectați opțiunea *Test*, programul va afișa consecutiv trei casete de mesaj fiecare cu șirul „Value for Property n ", unde n este egal cu 1, 2 sau 3 în funcție de caseta de dialog. Rețineți linia care apelează funcția *EnumProps*, arătată mai jos:

```
EnumProps(hWnd, (PROPENUMPROC) EnumPropsProc);
```

Amintiți-vă că operatorul de conversie (*PROPENUMPROC*) este o conversie explicită la un pointer de tip *PROPENUMPROC*. Acest pointer indică locația din memorie de unde începe funcția *EnumPropsProc*.

FUNCȚIILE CALLBACK

C/C++1285

În secțiunile precedente, ați folosit funcții callback pentru a realiza anumite obiective. Așa cum ați văzut, funcția callback este importantă în programarea Windows. Pentru a înțelege logic cum se încadrează funcția callback într-un apel de funcție, analizați diagrama logică de mai jos.

Programele dumneavoastră vor utiliza funcțiile callback în diverse moduri, însă de cele mai multe ori veți folosi funcțiile callback în apeluri de funcții API care generează un număr necunoscut de valori returnate. Veți întâlni frecvent funcții care execută activități callback, a căror denumire începe cu *Enum*, inclusiv *EnumFromFamilies*, *EnumWindows*, *EnumProps* etc.

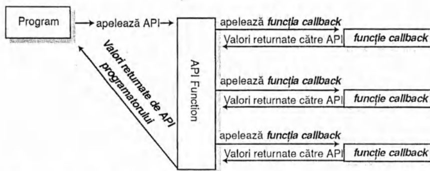


Figura 1285 *Modelul logic al prelucrării funcției callback.*

1286 FUNCȚIA MESSAGEBOX



În secțiunile 1283 și 1284 programul dumneavoastră a folosit funcția *MessageBox* pentru a afișa o casetă de dialog simplă, care necesită interacțiunea cu utilizatorul înainte ca programul să poată continua prelucrarea. În secțiunile următoare veți învăța cum să creați casete de dialog de multe tipuri, dar pentru o casetă simplă de dialog cu utilizatorul programul poate folosi funcția *MessageBox*. Funcția *MessageBox* produce, afișează și operează o casetă de mesaj. Caseta de mesaj conține un mesaj definit de aplicație, un titlu și orice combinație de pictograme și butoane cu apăsare predefinite. O casetă de mesaj nu poate afișa alte ferestre în interiorul său. Programul dumneavoastră va utiliza funcția *MessageBox* în următoarea formă generalizată:

```
int MessageBox (
    HWND hWnd;          //identificator pentru fereastra
    LPCTSTR lpText;     //adresa textului din caseta de mesaj
    LPCTSTR lpCaption;  //adresa cu titlul casetei de mesaj
    UINT uType;         //stilul casetei
);
```

Funcția *MessageBox* acceptă parametrii descriși în detaliu în Tabelul 1286.1.

Parametru	Descriere
hWnd	Identifică fereastra proprietar de care aparține caseta de mesaj. Dacă parametrul e NULL, caseta de mesaj nu are fereastră proprietar.
lpText	Pointer la un șir terminat cu NULL care conține mesajul care se va afișa.
lpCaption	Pointer la un șir terminat cu NULL folosit la titlul casetei. Dacă parametrul este NULL sistemul va utiliza implicit titlul *Error*.
uType	Specifică o serie de biți indicatori care determină conținutul și comportamentul casetei de dialog. Parametrul poate fi o combinație de indicatoare prezentate în următorul grup. Tabelul 1286.2 prezintă în detaliu indicatoare pentru butoanele utilizate de caseta de mesaj.

Tabelul 1286.1 *Parametrii funcției* ***MessageBox***.

Pe lângă textul casetei de mesaj stabilit prin parametrul *lpText* și titlul casetei stabilit prin *lpCaption*, programul va manipula de multe ori parametrul *uType* pentru a schimba butoanele și pictogramele care apar în casetă. Tabelul 1286.2 prezintă valorile posibile pentru a controla numărul și semnificația butoanelor din caseta de mesaj.

Valori	**Descriere**
MB_ABORTRETRYIGNORE	Caseta de mesaj conține trei butoane: *Abort, Retry* și *Ignore*.
MB_OK	Caseta de mesaj conține un buton: OK. MB_OK este valoarea implicită pentru parametrul uType.
MB_OKCANCEL	Caseta de mesaj conține două butoane: OK și Cancel.
MB_RETRYCANCEL	Caseta de mesaj conține două butoane: *Retry* și *Ignore*.
MB_YESNO	Caseta de mesaj conține două butoane: Yes și No.
MB_YESNOCANCEL	Caseta de mesaj conține trei butoane: *Yrs, No și Cancel*.

Tabelul 1286.2 *Valorile posibile ale parametrului **uType** pentru a controla numărul și semnificația butoanelor din caseta de mesaj.*

În plus față de controlul numărului și semnificației butoanelor din caseta de mesaj, programul dumneavoastră poate de asemenea controla dacă o pictogramă va apărea sau nu în caseta de mesaj. Valoarea implicită este MB_NOICON. Tabelul 1286.3 prezintă principalele valori ale parametrului *uType* pentru controlul apariției pictogramelor în caseta de mesaj.

Valori	**Descriere**
MB_ICONEXCLAMATION	În caseta de mesaj apare o pictogramă cu un semn de exclamare.
MB_NOICON	În caseta d emesaj nu apare nici o pictogramă.
MB_ICONASTERISK	Similar cu MB_ICONINFORMATION.
MB_ICONERROR	Similar cu MB_ICONHAND.
MB_ICONHAND	Similar cu MB_ICONSTOP.
MB_ICONINFORMATION	În caseta de mesaj apare o pictogramă cu litera i mic într-un cerc.
MB_ICONQUESTION	În caseta de mesaj apare o pictogramă cu un semn de întrebare.
MB_ICONSTOP	În caseta de mesaj apare o pictogramă cu un semn de stop.
MB_ICONWARNING	Similar cu MB_ICONEXCLAMATION.

Tabelul 1286.3 *Valorile posibile ale parametrului **uType** pentru controlul pictogramelor din caseta de mesaj.*

Funcția *MessageBox* acceptă numeroase constante suplimentare ale parametrului *uType*. Constantele listate în Tabelul 1286.2 și Tabelul 1286.3 sunt totuși cel mai frecvent folosite. Pentru celelalte constante, folosiți sistemul de help on-line al compilatorului.

Compact-discul care însoțește această carte include programul *Show_Mess.cpp*. Acest program generează un sunet de care veți afla în următoarea secțiune, apoi spune „Hello" utilizatorului și așteaptă să apese o tastă. Programul nu execută procese utile ca răspuns la selecțiile pentru care utilizatorul optează în caseta de mesaj, dar se pot realiza relativ ușor.

1287 FUNCȚIA MESSAGEBEEP



În secțiunea 1286 ați folosit funcția *MessageBox* pentru a genera o casetă de dialog simplă. Programul *Show_Mess.cpp* invocă funcția *MessageBeep* la crearea casetei de mesaj. Funcția *MessageBeep* redă un sunet de tip wave. Secțiunea *[sounds]* din registrul sistem identifică forma de undă a fiecărui tip de sunet. Programul va utiliza funcția *MessageBeep* cu următorul prototip:

```
BOOL MessageBeep (UINT uType) ;
```

Parametrul *uType* precizează tipul sunetului identificat de intrarea în secțiunea *[sounds]* din registrul sistem. Puteți încerca fiecare sunet din Windows folosind Control Panel Sounds din Windows dacă nu sunteți sigur de forma sunetului. Parametrul *uType* poate lua una din valorile prezentate în Tabelul 1287:

Valoare	Sunet
0xFFFFFFFF	Sunetul standard folosind difuzorul calculatorului
MB_ICONASTERISK	SystemAsterisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

Tabelul 1287 *Valorile posibile ale parametrului de sunet* ***uType***.

Programul dumneavoastră poate folosi funcția *MessageBeep* pentru a crea sunete simple, necesare atenționării utilizatorului asupra anumitor evenimente.

1288 DIN NOU DESPRE MESAJE



Așa cum ați învățat, administrarea mesajelor este „inima" care face ca sistemul Windows să funcționeze. Atât sistemul de operare, cât și aplicațiile care rulează pe acest sistem de operare generează mesaje despre fiecare eveniment care apare în Windows. Mesajele au o importanță fundamentală pentru valoarea sistemului Windows ca sistem de operare multi-tasking. Așa cum veți învăța, fiecare task (sau program) utilizează unul sau mai multe fire de execuție în cadrul sistemului de operare. Platforma Windows pe 32 de biți (Windows NT și Windows 95) întrețin un set separat de mesaje (o coadă de mesaje) pentru fiecare fir în funcțiune în sistemul de operare.

Windows generează mesaje pentru fiecare eveniment hardware care intervine, cum ar fi o apăsare de tastă sau un clic de mouse. Windows trece apoi fiecare mesaj în coada de mesaje corespunzătoare. Cu alte cuvinte, dacă utilizatorul execută clic cu mouse-ul, dar nu în cadrul aplicației dumneavoastră, aplicația nu va ști că utilizatorul a executat clic. Câteodată, sistemul va genera mai multe copii ale unui mesaj pe care le plasează simultan în mai multe cozi de mesaje. Figura 1288 prezintă o diagramă simplă a modului în care Windows prelucrează mesajele în cadrul mai multor cozi de mesaje.

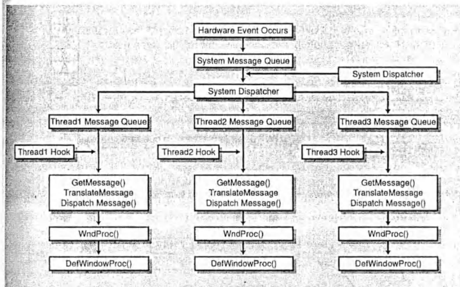


Figura 1288 *Windows prelucrează mesajele în cadrul mai multor cozi de mesaje.*

FLUXUL MESAJELOR

C/C++ 1289

Așa cum ați învățat, structura de mesaje a sistemului Windows este fundamentală pentru modul în care Windows gestionează mai multe taskuri în secvențe închise. Trebuie să înțelegeți fluxul de mesaje, nu numai la nivelul cozii de mesaje a programului, ci și a cozii de mesaje a programului din bucla de mesaje a programului dumneavoastră. Când Windows acceptă un mesaj hardware al calculatorului, el determină în sistemul intern cărei cozi de mesaje va trimite acel mesaj. După ce Windows trece mesajul în coada de mesaje a programului, programul la rândul lui prelucrează fiecare mesaj. De exemplu, uneori când introduceți un text pentru un procesor de text, se întâmplă să apăsați tastele mai rapid decât poate afișa monitorul. Programul însă, este capabil să mențină ceea ce tastați, chiar când ecranul încearcă să țină pasul, pentru că Windows stochează fiecare acționare a tastelor în coada de mesaje a programului, cum se vede în Figura 1289.1.

După ce Windows a plasat evenimentele de la tastatură în coada de mesaje, programul extrage mesaj cu mesaj din coada de mesaje, începând cu primul mesaj primit și continuând în ordine până la ultimul mesaj. După ce preia fiecare mesaj, programul folosește bucla de mesaje pentru a apela o funcție callback a mesajului, care prelucrează fiecare intrare în modul prezentat în Figura 1289.2

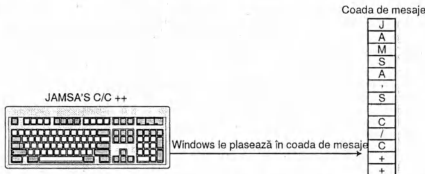


Figura 1289.1 *Windows prelucrează evenimentele da la tastatură și le plasează în coada de mesaje.*

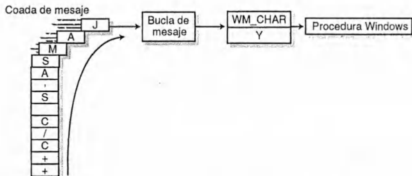


Figura 1289.2 *Bucla de mesaje a programului apelează funcția callback de prelucrare a mesajului.*

Așa cum ați învățat, programul dumneavoastră va testa ulterior valoarea din mesaje pentru a determina cum să răspundă la ele. Dacă, de exemplu, comanda este un eveniment de la tastatură, pe care utilizatorul vrea să îl plaseze într-un document curent al unui procesor de text, bucla de mesaje va trimite caracterul în fereastra curentă și îl va adăuga documentului din procesorul de text la locația curentă, în modul prezentat în Figura 1298.3.

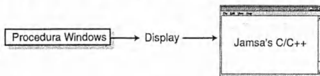


Figura 1298.3 *Funcția calback de prelucrare a mesajului inserează caracterul în documentul din procesorul de text.*

COMPONENTELE STRUCTURII MSG

C/C++1290

Așa cum ați învățat, mesajul Windows este un element de bază prin care creați programul dumneavoastră. De asemenea, ați învățat că Windows definește structura MSG în felul următor:

```
typedef struct tagMSG {
    HWND hwnd;              //identificator pentru fereastra
    UINTmessage             //ID mesaj
    WPARAM wParam;          //valoare wParam
    LPARAM lParam;          //valoare lParam
    DWORD time;             //nr. milisecunde de la startup
    POINT pt;               //coordonatele de ecran curente ale
                            //mouse-ului
} MSG;
```

Deși din diverse motive, fiecare din componentele structurii mesajului este importantă, veți utiliza mai frecvent în această carte două componente: *message* (pe care *WndProc* îl primește ca *uMsg*) și *wParam*. Când apelați funcția *DispatchMessage*, apelată întotdeauna în bucla de mesaje, funcția *DispatchMessage* trimite mesajul funcției *WndProc* al cărui antet îl veți declara în modul următor:

```
LRESULT CALLBACK WndProc (HWND hWnd, UINT uMsg, WPARAM wParam,
                          LPARAM lParam);
```

Funcția *DispatchMessage* transmite funcției *callback* numai următoarele componente membre al structurii *MSG*: *message*, *hWnd*, *wParam* și *lParam*. Așa cum ați învățat, funcția *callback* folosește atunci parametrul *uMsg* pentru determinarea inițială a tipului de mesaj. Dacă tipul de mesaj este o comandă Windows, fără parametrii, funcția *callback* execută prelucrarea corespunzătoare (de exemplu: *WM_DESTROY*). Pe de altă parte, dacă mesajul este *WM_COMMAND*, funcția *callback* trebuie mai întâi să testeze octetul mai puțin semnificativ al parametrului *wParam* pentru a determina ce comandă specifică a primit sistemul.

Așa cum ați învățat, veți defini constante de identificare pentru operațiile de tip comandă din programul dumneavoastră, în așa fel încât funcția *callback* să poată prelucra corect comenzile când le primește. De exemplu, majoritatea programelor pe care le-ați produs până acum folosesc identificatorul *IDM_TEST* pentru a determina când utilizatorul a selectat opțiunea Test din meniu. Pe măsură ce programul dumneavoastră câștigă în complexitate, veți folosi identificatori cu aproape toți membrii componenți ai unei ferestre, pentru a vă asigura că funcția *callback* poate prelucra corect selecțiile.

FUNCȚIA PEEKMESSAGE

C/C++1291

Așa cum ați învățat, programul dumneavoastră poate primi mesaje din coada de mesaje folosind fie funcția *GetMessage*, fie funcția *PeekMessage*. Funcția *PeekMessage* testează un mesaj în coada de mesaje a firului și, dacă îl găsește, îl plasează într-o structură specifică, prezentată mai jos:

```
BOOL PeekMessage(
    LPMSG lpMsg,            //pointer la structura mesajului
```

```
    HWND hWnd,              //identificator pentru fereastra
    UINT wMsgFilterMin,     //primul mesaj
    UINT wMsgFilterMax,     //ultimul mesaj
    UINT wRemoveMsg         //indicatoare de eliminare
);
```

Funcția *PeekMessage* acceptă parametrii prezentați în detaliu mai jos:

Parametru	Descriere
lpMsg	Pointer către o structură MSG care conține datele despre mesajul din coada de mesaje ale aplicației Windows.
hWnd	Identificator pentru fereastra care deține mesajele pe care programul dumneavoastră trebuie să le examineze.
wMsgFilterMin	Valoarea primului mesaj din intervalul de mesaje pe care programul dumneavoastră trebuie să le examineze.
wMsgFilterMax	Valoarea ultimului mesaj din intervalul de mesaje pe care programul dumneavoastră trebuie să le examineze.
wRemoveMsg	Specifică modul în care sunt gestionate mesajele programului. Acest parametru poate lua una dintre valorile specificate în Tabelul 1291.2.

Tabelul 1291.1 *Parametrii funcției* ***PeekMessage***.

Valoare	Descriere
PM_NOREMOVE	Mesajele nu sunt eliminate din coadă după prelucrarea din PeekMessage.
PM_REMOVE	Mesajele sunt eliminate din coadă după prelucrarea din PeekMessage.

Tabelul 1291.2 *Valorile parametrului* ***wRemoveMsg*** *al funcției* ***PeekMessage***.

Aveți opțiunea de a combina valoarea *PM_NOYIELD* fie cu *PM_NOREMOVE*, fie cu *PM_REMOVE*. Dar *PM_NOIYELD* nu are efect în aplicațiile Windows pe 32 de biți. El este definit în Win32 numai pentru a conferi compatibilitate cu aplicațiile scrise în versiunile anterioare de Windows, în care este folosit pentru a preveni oprirea taskului curent și transferul resurselor sistemului altui task. Aplicațiile Windows pe 32 de biți se execută întotdeauna simultan.

Spre deosebire de funcția *GetMessage*, funcția *PeekMessage* nu așteaptă ca Windows să plaseze un mesaj în coadă înainte de a reveni la locația apelantă. *PeekMessage* preia doar mesaje asociate cu fereastra identificată de parametrul *hWnd* sau cu oricare din descendenții ei, specificați de funcția *IsChild* și în intervalul valorilor specificate de parametrii *wMsgFilterMin* și *wMsgFiletrMax*. Dacă *hWnd* este NULL, funcția *PeekMessage*, preia mesajul pentru orice fereastră care aparține firului curent care face apelul. (*PeekMessage* nu preia mesaje pentru ferestre care aparțin altor fire). Dacă *hWnd* este –1, *PeekMessage* va returna mesaje numai cu valoarea NULL pentru *hWnd*. Dacă *wMsgFilterMax* și *wMsgFilterMin* sunt amândoi zero, *PeekMessage* va returna toate mesajele disponibile (adică, nu execută nici o filtrare a intervalului).

Puteți folosi constantele *WM_KEYFIRST* și *WM_KEYLAST* ca valori filtru pentru a prelua toate mesajele de la tastatură sau puteți folosi constantele *WM_MOUSEFIRST* și *WM_MOUSELAST* pentru a prelua toate mesajele mouse-ului.

Funcția *PeekMessage* de regulă nu elimină mesajele *WM_PAINT* din coadă. Mesajele *WM_PAINT* rămân în coada de mesaje până când sunt prelucrate de program. Dacă însă, mesajul *WM_PAINT* are o regiune actualizată cu valoarea NULL, *PeekMessage* o va elimina din coadă.

FUNCȚIA POSTMESSGE

C/C++1292

În secțiunile anterioare, ați învățat diverse modalități de preluare a mesajelor din coadă, de către programul dumneavoastră. Uneori veți dori să plasați mesaje în coada de mesaje a programului. Funcția *PostMessage* expediază mesajul în coada de mesaje asociată firului care a creat respectiva fereastră, apoi revine fără să aștepte ca firul să prelucreze mesajul. Apelul la funcțiile *GetMessage* sau *PeekMessage* determină preluarea mesajelor din coada de mesaje anterior expediată cu *PostMessage*. Veți folosi funcția *PostMessage* în cadrul programelor dumneavoastră cu următorul prototip:

```
BOOL PostMessage(
    HWND hWnd,          //identificator pentru fereastra destinatie
    UINT Msg,           //mesajul de expediat
    WPARAM wParam,      //parametrul primului mesaj
    LPARAM lParam       //parametrul celui de al doilea mesaj
);
```

Tabelul 1292.1 prezintă în detaliu parametrii acceptați de funcția *PostMessage*.

Parametru	Semnificație
hWnd	Identifică fereastra a cărei procedură fereastră trebuie să preia mesajul. Două valori au o semnificație deosebită. Detalii în tabelul 1292.2.
Msg	Specifică mesajul pe care PostMessage să-l pună în coadă.
wParam	Specifică informații suplimentare tipice mesajului.
lParam	Specifică informații suplimentare tipice mesajului.

Tabelul 1292.1 *Parametrii acceptați de funcția* ***PostMessage***.

Două valori ale parametrului *hWnd* au o semnificație specială pentru funcția *PostMessage*. Detalii în Tabelul 1292.2.

Valoare	Semnificație
HWND_BROADCAST	Mesajul este expediat tuturor ferestrelor din ierarhia superioară de ferestre a sistemului, inclusiv ferestre inactive sau invizibile fără proprietar, ferestre cu suprapunere și ferestre pop-up. Mesajul nu este expediat către ferestrele copil.
NULL	Funcția se comportă ca un apel la *PostThreadMessage* cu parametrul *dwThreadId* cu valoarea identificatorului firului curent.

Tabelul 1292.2 *Valorile speciale ale parametrului* ***bWnd*** *din funcția* ***PostMessage***.

Aplicațiile care trebuie să folosească *HWND_BROADCAST* pentru a comunica trebuie să folosească funcția *RegisterWindowMessage* pentru a obține mesajul unic pentru comunicarea între aplicații.

Dacă trimiteți un mesaj într-un interval mai jos de *WM_USER* către funcții de mesaj asincrone (*PostMessage*, *SendNotifyMessage* și *SendMessageCallback*), asigurați-vă că parametrii mesajelor nu includ pointeri. Altfel, funcțiile vor reveni înainte ca firul receptor să fi avut șansa de a prelucra mesajul, iar expeditorul va șterge memoria înainte ca programul să o folosească.

1293 FUNCȚIA SENDMESSAGE



În secțiunea 1282 ați învățat cum să utilizați funcția *PostMessage* pentru a plasa un mesaj în coada de mesaje a unui fir. Așa cum ați învățat, *PostMessage* returnează controlul către programul apelant fără să aștepte un răspuns de la firul care primește mesajul. Uneori, programul poate să necesite un răspuns de la firul care primește mesajul, înainte ca procesul să poată continua în firul apelant. Puteți folosi funcția *SendMessage* pentru a controla când prelucrarea aplicației returnează controlul către aplicația apelantă. Funcția *SendMessage* trimite mesajul specificat uneia sau mai multor ferestre. Funcția *SendMessage* apelează procedura fereastră pentru respectiva fereastră și nu returnează controlul către programul apelant până când procedura fereastră nu prelucrează mesajul. Funcția *PostMessage*, dimpotrivă, expediază un mesaj către coada de mesaje a firului și returnează controlul imediat. Programul dumneavoastră va utiliza funcția *SendMessage* cu prototipul de mai jos:

```
LRESULT SendMessage(
HWND hWnd,         // identificator pentru fereastra destinatie
UINT Msg,          // mesajul de trimis
WPARAM wParam,     // primul parametru mesaj
LPARAM lParam      // al doilea parametru mesaj
);
```

Parametrii funcției *SendMessage* sunt identici cu parametrii funcției *PostMessage*. Dacă însă stabiliți parametrul *hWnd* ca *HWND_BROADCAST*, programul va trimite mesajul către toate ferestrele de la nivelul de vârf din sistem, inclusiv ferestre fără proprietar, invizibile sau dezactivate, ferestre suprapuse și ferestre pop-up. Programul nu va trimite însă mesajul către ferestrele copil.

Aplicațiile care trebuie să utilizeze *HWND_BROADCAST* pentru a comunica trebuie să folosească funcția *RegisterWindowMessage* pentru a obține un mesaj unic pentru comunicări între aplicații.

Dacă firul apelant a creat fereastra respectivă, Windows apelează procedura fereastră imediat ca subrutină. Dacă un fir diferit a creat fereastra respectivă, Windows se comută la firul acelei ferestre și apelează procedura fereastră corespunzătoare. Windows prelucrează mesajele trimise între fire numai când firul primitor execută codul de preluare a mesajului. Windows blochează firul care expediază mesajul, până când firul receptor prelucrează acel mesaj.

1294 UTILIZAREA FUNCȚIEI REPLYMESSAGE



În secțiunile anterioare ați învățat cum să folosiți funcțiile *PostMessage* și *SendMessage* în cadrul programelor dvs. Programul va folosi funcția *ReplyMessage* pentru a răspunde unui mesaj trimis cu funcția *SendMessage*, fără să redea controlul funcției care a apelat *SendMessage*, ca mai jos:

```
BOOL ReplyMessage(
  LRESULT lResult // rezultat specific mesajului
);
```

Apelând funcția *ReplyMessage*, procedura fereastră care primește mesajul permite firului care a apelat funcția *SendMessage* să ruleze ca și când firul ce a primit mesajul a redat controlul. Firul care apelează funcția *ReplyMessage* continuă să ruleze.

Dacă programul nu a transmis mesajul prin *SendMessage* sau dacă același fir a trimis mesajul, *ReplyMessage* nu are nici un efect.

MESAJE DE INTERCEPTARE

C/C++1295

Așa cum ați învățat, mesajele stau la baza programelor de administrare a sistemului de operare Windows. Așa cum, de asemenea, ați învățat, puteți utiliza funcții API diferite pentru a transmite mesajele între două fire. Uneori însă, veți dori ca un anumit program să intercepteze toate mesajele pe care Windows le trimite altui program. *Interceptorii de mesaje* vă permit să captați mesajele adresate unui program, într-un program diferit. Programul de interceptare poate acționa asupra mesajelor pe care le interceptează, le poate modifica sau chiar opri. Veți intercepta deseori mesajele din bibliotecile cu legare dinamică – DLL, un tip special de fișier suport, care furnizează servicii suplimentare de prelucrare a mesajelor, fără să încetinească programul principal. Domeniul de valabilitate al unui interceptor depinde de tipul de interceptare. Puteți fixa unii interceptori numai la domeniul de valabilitate al sistemului, pe când alții îi veți stabili numai pentru un anumit fir, așa cum arătăm în următorul tabel. Tabelul 1295 listează tipurile de interceptori, în ordinea în care sistemul de operare face interceptarea.

Interceptor	Domeniu de valabilitate
WH_CALLWNDPROC	Fir sau sistem
WH_CALLWNDPROCRET	Fir sau sistem
WH_CBT	Fir sau sistem
WH_DEBUG	Fir sau sistem
WH_GETMESSAGE	Fir sau sistem
WH_JOURNALPLAYBACK	Numai sistem
WH_JOURNALRECORD	Numai sistem
WH_KEYBOARD	Fir sau sistem
WH_MOUSE	Fir sau sistem
WH_MSGFILTER	Fir sau sistem
WH_SHELL	Fir sau sistem
WH_SYSMSGFILTER	Numai sistem

Tabelul 1295 *Domeniul de valabilitate a interceptorilor de sistem.*

Pentru un anumit tip de interceptor, Windows apelează mai întâi interceptorii de fir, apoi pe cei de sistem.

1296 UTILIZAREA FUNCȚIEI SETWINDOWSHOOKEX

C/C++

În secțiunea 1295 ați învățat despre interceptorii de mesaj pe care programul dvs. îi poate utiliza pentru a lansa interceptori pentru alte aplicații. Funcția *SetWindowsHookEx* instalează o procedură de interceptare definită la nivel de aplicație, într-un lanț de interceptori. Programul dumneavoastră poate utiliza procedurile de interceptare pentru a monitoriza sistemul pentru anumite tipuri de evenimente. Când instalați o procedură de interceptare, Windows asociază evenimentele monitorizate de procedura de interceptare fie cu un anumit fir, fie cu toate firele din sistem. Veți utiliza funcția *SetWindowsHookEx* în cadrul programelor dumneavoastră cu următorul prototip:

```
HHOOK SetWindowsHookEx(
    int idHook,        // tipul de interceptor pt instalare
    HOOKPROC lpfn,     // adresa procedurii de interceptare
    HINSTANCE hMod,    // identificator pentru instanta aplicatiei
    DWORD dwThreadId   // identitatea firului pentru care se
                       // instaleaza interceptorul
);
```

Parametrul *idHook* precizează tipul procedurii de interceptare de instalat în program. Acest parametru acceptă una dintre valorile listate în Tabelul 1296. Parametrul *lpfn* indică procedura de interceptare. Dacă parametrul *dwThreadId* este zero sau menționează identificatorul unui fir creat de un alt proces, atunci parametrul *lpfn* trebuie să indice o procedură de interceptare dintr-o bibliotecă cu legătură dinamică (DLL). Altfel, *lpfn* poate indica o procedură de interceptare din codul asociat cu procesul curent. Parametrul *hMod* identifică biblioteca cu legătură dinamică care conține procedura de interceptare creată de procesul curent. Trebuie stabilită valoarea NULL pentru *hMod* dacă parametrul *dwThreadId* specifică un fir creat de procesul curent și dacă procedura de interceptare este în codul asociat procesului curent. Parametrul *dwThreadId* precizează identificatorul firului la care programul dumneavoastră a asociat procedura de interceptare. Dacă parametrul este zero, programul va asocia procedura de interceptare la toate firele existente.

Așa cum probabil vă așteptați, puteți stabili o varietate de tipuri de interceptori. Veți determina ce tipuri de interceptori vor fi stabiliți de sistem cu valoarea pe care o plasați în parametrul *idHook*. Tabelul 1296 prezintă în amănunt valorile posibile ale parametrului *idHook*.

Valoare	Descriere
WH_CALLWNDPROC	Instalează o procedură de interceptare care monitorizează mesajele înainte ca sistemul să le trimită la procedura fereastră de destinație.
WH_CALLWNDPROCRET	Instalează o procedură de interceptare care monitorizează mesajele după ce ele au fost prelucrate de procedura fereastră de destinație.
WH_CBT	Instalează o procedură de interceptare care primește notificări necesare unei aplicații CBT (computer-based training – instruire asistată de calculator).

Valoare	Descriere
WH_DEBUG	Instalează o procedură utilă pentru depanarea altor proceduri de interceptare.
WH_GETMESSAGE	Instalează o procedură de interceptare care monitorizează mesajele expediate unei cozi de mesaje.
WH_JOURNALPLAYBACK	Instalează o procedură de interceptare care expediază mesaje anterior înregistrate cu procedura de interceptare *WH_JOURNALRECORD.*
WH_JOURNALRECORD	Instalează o procedură de interceptare care înregistrează mesajele de intrare expediate cozii de mesaje a sistemului. Această interceptare este utilă pentru înregistrarea macrocomenzilor.
WH_KEYBOARD	Instalează o procedură de interceptare care monitorizează mesajele de la tastatură.
WH_MOUSE	Instalează o procedură de interceptare care monitorizează mesajele de la mouse.
WH_MSGFILTER	Instalează o procedură de interceptare care monitorizează mesajele generate ca rezultat al unui eveniment de intrare într-o casetă de dialog, casetă de mesaj, meniu sau bară de defilare.
WH_SHELL	Instalează o procedură de interceptare care primește notificări utile aplicațiilor de tip *shell.*
WH_SYSMSGFILTER	Instalează o procedură de interceptare care monitorizează mesajele generate ca rezultat al unui eveniment de intrare într-o casetă de dialog, casetă de mesaj, meniu sau bară de defilare. Procedura de interceptare monitorizează aceste mesaje pentru toate aplicațiile din sistem.

Tabelul 1296 *Valorile posibile ale parametrului* ***idHook***.

Dacă parametrul *hMod* este NULL și parametrul *dwThreadId* este zero sau specifică identificatorul unui fir produs de un alt proces, poate apărea o eroare.

Apelarea funcției *CallNextHookEx* pentru înlănțuirea cu următoarea procedură de interceptare este opțională. Însă, așa cum veți învăța, dacă nu folosiți funcția *CallNextHookEx*, alte aplicații care au instalat anterior interceptori nu vor primi notificări de interceptare și pot în consecință să se comporte incorect. Va trebui să apelați *CallNextHookEx* dacă nu trebuie să preveniți în mod absolut ca alte aplicații să vadă o notificare.

Înainte de a se termina, o aplicație trebuie să apeleze funcția *UnhookWindowsHookEx* pentru a elibera resursele sistemului, asociate cu interceptorul.

Interceptorii de sistem sunt resurse partajate, iar instalarea unuia afectează toate aplicațiile. Toate funcțiile de interceptare trebuie să fie în biblioteci. Va trebui să restrângeți interceptorii de sistem la aplicații cu obiective speciale sau să îi utilizați în dezvoltarea de aplicații și în timpul depanării lor. Bibliotecile care nu mai au nevoie de interceptori trebuie să șteargă procedura de interceptare.

1297 FUNCȚIA EXITWINDOWEX



În secțiunile precedente ați învățat despre utilizarea mesajelor în programele dumneavoastră. Veți găsi o mică diferență în utilizarea mesajelor în funcția *ExitWindowsEx*. Funcția *ExitWindowsEx* fie închide accesul la sistem, închide sistemul, fie închide și repornește sistemul. Programele dumneavoastră vor utiliza funcția *ExitWindowsEx* cu prototipul de mai jos:

```
BOOL ExitWindowsEx(
    UINT uFlags:            //operatia de inchidere
    DWORD dwReserved:       //rezervat
):
```

În cadrul funcției *ExitWindowsEx*, parametrul *uFlags* precizează tipul de închidere. Parametrul *uFlags* trebuie să fie o combinație a valorilor descrise în Tabelul 1297. Windows ignoră în prezent parametrul *dwReserved*, el fiind rezervat pentru dezvoltări ulterioare. Tabelul 1297 prezintă posibilele valori ale parametrului *uFlags*:

Valoare	Descriere
EWX_FORCE	Forțează terminarea procesului. Când programul activează acest indicator, Windows nu trimite mesajele *WM_QUERYENDSESSION* și *WM_ENDSESSION* către aplicațiile care rulează la acel moment în sistem. Aceasta poate cauza pierderea datelor din aplicații. Deci acest indicator va fi utilizat numai în caz de urgență.
EWX_LOGOFF	Închide toate procesele care rulează în contextul de securitate al procesului care a apelat funcția *ExitWindowsEx*. Apoi este oprit accesul utilizatorului la sistem.
EXW_POWEROFF	Închide sistemul și oprește alimentarea de la rețea. Sistemul trebuie să accepte oprirea alimentării de la rețea.
***Observație:** În Windows NT, procesul apelant trebuie să aibă privilegiul SE_SHUTDOWN_NAME. Windows 95 nu acceptă și nu necesită privilegii de securitate.*	
EWX_REBOOT	Închide sistemul apoi repornește sistemul.
***Observație:** În Windows NT, procesul apelant trebuie să aibă privilegiul SE_SHUTDOWN_NAME.*	
EWX_SHUTDOWN	Închide sistemul la un moment în care este sigură oprirea alimentării de la rețea. Sistemul salvează toate bufferele de fișier pe disc și încheie toate procesele care rulează.

***Tabelul 1297** Valorile posibile ale parametrului **uFlags**.*

Funcția *ExitWindowsEx* revine imediat ce a inițializat închiderea. Încetarea accesului și închiderea sistemului se efectuează asincron (adică programele dumneavoastră vor efectua prelucrări suplimentare în timpul procesului de închidere). În timpul operațiilor de încetare a accesului și închidere, sistemul oferă aplicațiilor care se închid un anumit interval de timp pentru a răspunde cererii de închidere. Dacă timpul expiră, Windows va afișa o casetă de dialog care permite utilizatorului să închidă forțat aplicația, să repete acțiunea de închidere sau să anuleze cererea de închidere. Dacă precizați valoarea *EWX_FORCE*, Windows va forța închiderea aplicațiilor și nu va afișa caseta de dialog pentru utilizator.

Funcția *ExitWindowsEx* trimite către *procesele de consolă*, un mesaj de notificare separat – *CTRL_SHUTDOWN_EVENT* sau *CTRL_LOGOFF_EVENT*, în funcție de situație. Un proces de consolă directează aceste mesaje către funcțiile sale *HandlerRoutine*, care apelează funcția *SetConsoleCtrlHandler* pentru adăugare și ștergere. *ExitWindowsEx* trimite aceste mesaje de notificare în mod asincron; în consecință o anumită aplicație nu poate presupune că funcțiile *HandlerRoutine* gestionează mesajele de notificare ale consolei când un apel la *ExitWindowsEx* returnează controlul.

Observație: *Pentru a închide sau a reporni un sistem Windows NT, procesul apelant trebuie să utilizeze funcția* ***AdjustTokenPrivileges*** *pentru a institui privilegiul SE_SHUTDOWN_NAME. Procesele de consolă rulează numai pe serverul Windows NT. Windows 95 nu acceptă sau nu necesită privilegii de securitate.*

TIPURILE DE MENIU C/C++1298

Așa cum ați văzut în secțiunile precedente, o componentă primară a arhitecturii Windows NT și Windows 95 este *bara de meniu*. Bara de meniu este în mod esențial un container pentru resursele de meniu. Așa cum ați văzut, resursele de meniu permit programelor dumneavoastră să primească introduceri de text și evenimente de mouse de la utilizator, în cadrul programului. De obicei, meniurile se grupează după anumite categorii și oferă utilizatorului multiple opțiuni de selectare din aceste categorii.

Programele Windows utilizează două tipuri de meniu: *meniu principal (top-level menu* sau *main menu)* și *meniu pop-up (derulant)*. Meniul principal reprezintă un grup de comenzi vizibile permanent în bara de meniu a ferestrei, dacă presupunem că programul posedă un meniu. Pentru programele mai complexe, nu poate fi întotdeauna posibil sau practic să puneți toate opțiunile de meniu, necesare programului, în bara de meniu. În aceste cazuri, programul dumneavoastră poate folosi meniurile pop-up (deseori numite sub-meniuri sau meniuri *pull-down* sau *drop-down*). Când selectați o opțiune din bara de meniu principal, Windows de asemenea desfășoară un meniu pop-up. Figura 1298 prezintă un meniu principal simplu cu un sub-meniu.

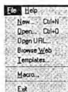


Figura 1298 *Un meniu principal simplu cu un sub-meniu atașat.*

Observație: *Nu confundați meniurile pop-up din contextul API Windows cu meniurile sensibile la context (context sensitive). Un meniu sensibil la context este un tip special de meniu, despre care veți învăța mai mult în secțiunile următoare.*

STRUCTURA MENIURILOR C/C++1299

Așa cum ați învățat, Windows își compune meniul program din una sau mai multe selecții dintr-un meniu principal, una sau mai multe selecții din cadrul fiecărei selecții din meniul

principal (numite opțiuni în această carte) și una sau mai multe selecții din fiecare selecție (numite sub-opțiuni aici). De exemplu, Figura 1299 prezintă structura arborescentă simplă a unor grupuri de meniuri dintr-un program simplu. Observați cum în ierarhia meniului, meniurile principale au un număr variabil de opțiuni care, în anumite cazuri, au la rândul lor sub-opțiuni, cum reiese din figura de mai jos.

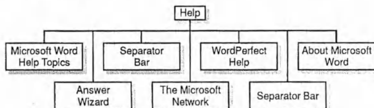


Figura 1299.1 *Structura ierarhică a meniurilor unui program.*

În plus, este important de observat că, într-o structură reușită de meniu, există un singur mod de a ajunge la orice opțiune sau sub-opțiune dată. Menținerea unei ierarhii stricte a meniului, evită confuzia între sub-opțiuni în cadrul meniurilor multiple. Când utilizați o structură de meniu strict ierarhică, meniul programului va trebui să arate structural identic cu diagrama din Figura 1299.1, cum rezultă din Figura 1299.2.

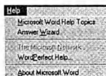


Figura 1299.2 *Meniul propriu-zis al programului.*

1300 CREAREA UNUI MENIU ÎN FIȘIERUL RESURSĂ C/C++

Așa cum ați învățat de-a lungul acestei cărți, cel mai adesea veți crea meniuri în cadrul fișierelor de resurse. Așa cum am menționat în multe secțiuni anterioare, cele mai multe pachete de dezvoltare C++ pentru Windows vă vor permite să proiectați meniuri prin metode simple de manipulare a mouse-ului. Este important însă să înțelegem modalitatea în care programele dumneavoastră creează meniuri, în cazul în care va trebui să le realizați manual. Următoarea listă, de exemplu, prezintă construcția cea mai obișnuită a unui meniu într-un fișier resursă.

```
MYAPP MENU DISCARDABLE
BEGIN
   POPUP "&File"
   BEGIN
     MENUITEM "E&xit", IDM_EXIT
   END
   POPUP "&Test"
```

```
    BEGIN
      MENUITEM "Item &1", IDM_ITEM1
      MENUITEM "Item &2", IDM_ITEM2
      MENUITEM "Item &3", IDM_ITEM3
    END
    POPUP "&Help"
    BEGIN
      MENUITEM "&About My Application...", IDM_ABOUT
    END
  END
```

In următoarele câteva secțiuni, veți studia în detaliu definirea meniului pentru fișierul cu resurse, precum și modul de utilizare a definiției în cadrul programelor. În secțiunea 1301 veți studia modul în care fișierele cu resurse folosesc descriptorii *POPUP* și *MENUITEM*. Este important de reținut descriptorul *DISCARDABLE* din declarația de resurse de mai sus. Descriptorul *DISCARDABLE* anunță editorul de legături al compilatorului că ar trebui să elimine informația inițială a resursei despre meniu după ce programul înregistrează meniul în clasa ferestrei. Veți utiliza aproape întotdeauna descriptorul *DISCARDABLE* în definițiile meniului pentru a economisi spațiu de memorie și a îmbunătăți viteza de prelucrare a programului.

DECRIPTORII POPUP ȘI MENUITEM

C/C++ 1301

După cum ați văzut în secțiunea 1300, descrierea meniului pentru fișierul de resurse începe cu instrucțiunea *BEGIN* și se încheie cu instrucțiunea *END*. În descrierea unui meniu, pot exista oricâte elemente *POPUP* și *MENUITEM*. Trebuie totuși evitată crearea prea multor elemente de meniu primare, deoarece e posibil să nu aibă loc toate în fereastră, atunci când fereastra are dimensiuni normale.

În descrierea de meniu, descriptorul *POPUP* indică nivelul de sus al meniului. De exemplu, elementul *Test!* este în partea de sus a unui meniu din programul secțiunii 1300. Utilizarea ampersandului (&) în șir indică tasta ce va fi utilizată pentru comanda rapidă în loc de mouse, pentru accesul la elementul meniului. Programatorii spun adesea despre comenzile rapide că sunt „scurtături". În cazul maniului *Test!*, comanda rapidă este ALT+K.

Fișierul de resurse folosește descriptorul *MENUITEM* pentru a lista fiecare element din cadrul meniului. De exemplu, descrierea meniului *Test!* din secțiunea 1300 conține trei elemente de meniu: Item #1, Item #2 și Item #3. Remarcați că fiecare element de meniu include o referință constantă. În cazul elementului 1, referința constantă este *IDM_ITEM1*. În rutina *WndProc*, programul va capta mai întâi mesajul *WM_COMMAND*. După ce programul captează mesajul *WM_COMMAND*, cuvântul mai puțin semnificativ al valorii *wParam* pe care o primește *WndProc* va conține identificatorul de constantă pentru elementul de meniu selectat de utilizator. De exemplu, pentru a prelucra selectarea opțiunii *IDM_ITEM1*, programul va folosi două instrucțiuni *switch*. Prima instrucțiune va verifica dacă mesajul este de tipul *WM_COMMAND*. A doua instrucțiune va verifica dacă valoarea din *wParam* reprezintă elementul de meniu Item #1 selectat de utilizator:

```
switch(uMsg)
  {
    case WM_COMMAND:
      switch(LOWORD(wParam))
      {
        case ITEM1:
          //unele prelucrari
        case ITEM2:
          //alte prelucrari
```

1302 ADĂUGAREA UNUI MENIU ÎN FEREASTRA APLICAȚIEI

C/C++

Așa cum ați învățat, definirea unui meniu în fișierul resursă (.RC) nu duce automat la utilizarea meniului sau atașarea lui la fereastra aplicației. În mod obișnuit, veți atașa meniul aplicației definiției clasei fereastră în funcția *WinMain*, înainte de a invoca funcția *RegisterClass*, cum ați procedat în secțiunile anterioare. Rețineți că fixați elementul *lpszMenuName* în structurile *WNDCLASS* și *WNDCLASSEX* pentru a realiza atașarea de tip legare timpurie (early binding). Funcția *RegisterClass* asociază în acel moment numele meniului cu oricare fereastră produsă mai târziu de către clasă.

Puteți, de asemenea, utiliza funcțiile *CreateWindow* și *LoadMenu* pentru a atașa un meniu la o fereastră. Când invocați funcția *CreateWindow*, puteți stabili parametrul *hMenu* la valoarea returnată de funcția *LoadMenu*. Atunci veți apela *LoadMenu* cu numele meniului din fișierul resursă, ca parametru al funcției, ca mai jos:

```
if (LWORD(wParam) == IDM_MEN)
   hNewMenu = LoadMenu(hInst, "NEWMENU");
else
   hNewMenu = LoadMenu(hInst, "OLDMENU");
```

În fine, puteți utiliza funcțiile *SetMenu* și *LoadMenu* pentru a atașa un meniu unei ferestre după ce ați creat fereastra. Din nou, veți invoca funcția *SetMenu* cu parametrul indicator *hMenu*. *SetMenu* asociază meniul cu indicatorul *hMenu* cu fereastra aplicației care este deschisă în acel moment, ca mai jos:

```
SetMenu(hWnd, hNewMenu);
```

1303 SCHIMBAREA MENIURILOR ÎN APLICAȚIE

C/C++

În secțiunile precedente ați învățat cum pot fi utilizate în programele dumneavoastră o varietate de metode de a atașa un meniu la o aplicație. Totuși, de-a lungul proiectării programului Windows, veți dezvolta aplicații mai complexe care de regulă reclamă schimbări de meniu în timpul executării aplicației. Intrefața Win32 API pune la dispoziție un set extensiv de funcții pe care le poate utiliza programul dumneavoastră pentru a schimba meniul în timpul derulării aplicației. Pe scurt, aceste funcții vă permit să schimbați fiecare aspect al meniului, după ce l-ați atașat la o fereastră.

Cele mai obișnuite schimbări de meniu pe care le va face programul este să schimbe textele afișate de articolele de meniu, să plaseze sau să elimine semne de validare, să activeze sau să dezactiveze articole de meniu, să șteargă sau să adauge articole de meniu. Funcția *ModifyMenu* vă permite să executați mai multe astfel de operații dintr-un singur apel la funcție. Pe de altă parte, puteți folosi funcții specifice, cum ar fi *DeleteMenu* sau *CheckMenuItem* pentru a modifica articolele din meniu. Compact-discul conține programul *Delete_Items.cpp* care conține comenzi de meniu pentru a permite utilizatorului să adauge sau să șteargă articole dintr-un meniu.

MESAJELE GENERATE DE MENIU

C/C++1304

Așa cum ați învățat, de fiecare dată când un utilizator selectează un articol de meniu în programul dumneavoastră, Windows va trimite mesajul *WM_COMMAND* în bucla de mesaje a programului. În cadrul buclei de mesaje, programul dumneavoastră trebuie să testeze cuvântul mai puțin semnificativ al valorii *wParam* de tip *DWORD* pentru a determina articolul din meniu selectat de utilizator.

Ca regulă, *WM_COMMAND* este singurul mesaj pe care Windows îl trimite programului dumneavoastră ca rezultat al selecției utilizatorului. Dacă, de exemplu, utilizatorul selectează un articol din meniul sistem, Windows va trimite în schimb mesajul *WM_SYSCOMMAND* (pe care îl veți trata în procedura *DefWindowProc*). Pe de altă parte, aplicația dumneavoastră va cere buclei de mesaje să trateze mesajele *WM_INITMENU* și *WM_INITMENUPOPUP* – trimise ferestrei de către sistem, imediat înainte ca sistemul să activeze un meniu (fie un meniu principal, fie unul derulant, în funcție de mesaj). Capturarea mesajelor *WM_INITMENU* și *WM_INITMENUPOPUP* permite aplicației dumneavoastră să schimbe unul sau mai multe meniuri, în cazul în care sunt necesare schimbări imediat ce Windows afișează meniul aplicației. Meniul va trimite, de asemenea, mesajul *WM_MENUSELECT* de fiecare dată când un utilizator selectează un articol din meniu. Mesajul *WM_MENUSELECT* este mai puternic și mai flexibil decât mesajul *WM_COMMAND*, pentru că Windows îl generează și atunci când articolul de meniu este în acel moment dezactivat. Veți utiliza însă în mod obișnuit numai mesajul *WM_MENUSELECT* pentru a afișa meniul *help* sensibil la context.

FUNCȚIA LOADMENU

C/C++1305

Așa cum ați învățat în secțiunea 1302, programul dumneavoastră poate utiliza funcția *LoadMenu* pentru a încărca un meniu pe care l-ați definit anterior în fișierul resursă. De regulă, veți urma fie invocarea lui *LoadMenu* cu apel la *SetMenu*, fie veți apela *LoadMenu* în cadrul invocării funcției *CreateWindow*. Funcția *LoadMenu* încarcă resursa de meniu pe care parametrul *lpMenuName* o specifică din fișierul executabil (.exe) asociat de Windows cu instanța aplicației. Veți utiliza funcția *LoadMenu* în programul dumneavoastră cu prototipul prezentat mai jos:

```
HMENU LoadMenu (
   HINSTANCE hInstance,   //identificator pentru instanta
                          //aplicatiei
   LPCTSTR lpMenuName     //sir cu numele meniului sau
                          //identificator de resursa pentru meniu
);
```

LoadMenu returnează un indicator *HMENU* și acceptă ca parametri identificatorul pentru instanța modulului care conține resursa de meniu pe care *LoadMenu* urmează să o încarce și un pointer la un șir de caractere terminat în NULL cu numele resursei de meniu. În loc să utilizați un pointer la un nume de meniu, puteți folosi un *DWORD* ca al doilea parametru (pointerul la numele meniului). În acest caz, *DWORD* va conține *identificatorul de resursă* cu identificatorul real în *cuvântul* inferior și zero în *cuvântul* superior. Pentru a crea valoarea *DWORD*, spre deosebire de pointerul la șirul constantă, utilizați macroinstrucțiunea *MAKEINTRESOURCE*.

Pentru a înțelege mai bine prelucrarea efectuată de *LoadMenu*, să analizăm fragmentul de cod care urmează din programul *2_menus*, care interschimbă două meniuri, în funcție de selecția utilizatorului. Atât fișierul resursă, cât și fișierul program diferă de corespondentele lor din fișierul *generic*. Fișierul *2_menus.rc* include următoarele definiții ale meniului:

```
OLDMENU MENU DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "E&xit",            IDM_EXIT
  END
  MENUITEM "&New Menu!"          IDM_NEW
END
NEWMENU MENU DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "E&xit",            IDM_EXIT
  END
  MENUITEM "&Old Menu!"          IDM_OLD
END
```

Prima declarație produce meniul *OLDMENU* cu opțiunea *New Menu!* Când utilizatorul selectează opțiunea *New Menu!*, programul va schimba meniul din cadrul ferestrei program, din meniul *OLDMENU* în meniul *NEWMENU*. Așa cum vă așteptați, procesul aplicației care a schimbat meniul aplicației din *OLDMENU* în *NEWMENU* se află în funcția *WndProc* din fișierul *2_menus.cpp*.

Așa cum veți vedea, programul testează valoarea constantei conținută de cuvântul inferior al valorii lui *wParam*. În funcție de rezultat, programul încarcă celălalt meniu (cu alte cuvinte, dacă rezultatul indică faptul că programul afișează la acel moment resursa *OLDMENU*, programul va afișa resursa *NEWMENU* și viceversa). Programul va folosi atunci funcția *SetMenu* pentru a comuta între cele două funcții. În final, programul invocă funcția *DrawMenuBar*, după ce termină comutarea, ceea ce asigură faptul că Windows retrasează meniul nou încărcat.

Observație: *Programul dumneavoastră trebuie să utilizeze funcția* ***DestroyMenu*** *înainte ca aplicația să se închidă, să distrugă meniul și să elibereze memoria ocupată de meniul încărcat.*

UTILIZAREA FUNCȚIEI MODIFYMENU

C/C++ 1306

Așa cum ați învățat în secțiunea 1303, programul dumneavoastră poate utiliza funcția *ModifyMenu* pentru a realiza o varietate de schimbări ale meniurilor, după ce programul le atașează unei ferestre. Funcția *ModifyMenu* schimbă un articol existent în meniu. Programul dumneavoastră poate utiliza funcția *ModifyMenu* pentru a arăta conținutul, aspectul și comportamentul fiecărui articol din cadrul unui meniu. Veți utiliza funcția *ModifyMenu*, în programul dumneavoastră cu prototipul de mai jos:

```
BOOL ModifyMenu(
    HMENU hMnu,           // identificator pentru meniu
    UINT uPosition,       // articol de modificat in meniu
    UINT uFlags,          // indicatoare pentru articole meniu
    UINT uIDNewItem,      // identificator articol meniu sau meniu
                          // derulant
    LPCTSTR lpNewItem     //continut articol meniu nou
);
```

Funcția *ModifyMenu* acceptă parametrii detaliați în Tabelul 1306.1.

Parametru	Descriere
hMnu	Identifică meniul ce va fi schimbat.
uPosition	Precizează articolul de meniu care urmează a fi schimbat, determinat de parametrul *uFlags*.
uFlags	Specifică indicatoarele care controlează interpretarea parametrului *uPosition* și conținutul, aspectul și comportamentul articolului de meniu. Acest parametru trebuie să fie o combinație a uneia din valorile cerute prezentate în Tabelul 1306.2, cu cel puțin una din valorile detaliate în Tabelul 1306.3.
uIDNewItem	Specifică fie identificatorul articolului modificat de meniu, fie identificatorul meniului derulant sau submeniu, în cazul în care parametrul *uFlags* are fixat indicatorul pe *MF_POPUP*.
lpNewItem	Indică spre conținutul articolului modificat din meniu. Interpretarea acestui parametru depinde de eventualitatea ca parametrul *uFlags* care poate să conțină valorile *MF_BITMAP*, *MF_OWNERDRAW* sau *MF_STRING*.

Tabelul 1306.1 *Parametrii acceptați de funcția* ***ModifyMenu***.

Așa cum arată Tabelul 1302.1, trebuie să treceți o valoare în parametrul *uFlags*. Valoarea lui *uFlags* trebuie să fie o combinație a unora din valorile listate în Tabelul 1306.2 și una sau mai multe din valorile listate în Tabelul 1306.3.

Valoare	Semnificație
MF_BYCOMMAND	Indică faptul că parametrul *uPosition* dă identificatorul articolului de meniu. Indicatorul este implicit *MF_BYCOMMAND* în cazul în care nici *MF_BYCOMMAND*, nici *MF_BYPOSITION* nu sunt specificate.
MF_BYPOSITION	Indică faptul că parametrul *uPosition* dă poziția relativă la zero a articolului de meniu.

Tabelul 1306.2 *Valorile posibile ale parametrului* ***uFlags***.

În plus față de valoarea cerută de parametrul *uFlags*, trebuie de asemenea să utilizați operatorul SAU (OR) pe bit pentru a atribui una sau mai multe valori ale articolelor de meniu detaliate în Tabelul 1306.3.

Valoare	Semnificație
MF_BITMAP	Conține un identificator de bitmap.
MF_OWNERDRAW	Conține o valoare de 32 de biți furnizată de aplicație pe care Windows o utilizează pentru a menține date suplimentare relativ la articolul de meniu. Valoarea este în membrul *itemData* al structurii indicate de parametrul *lparam* al mesajelor *WM_MEASUREITEM* sau *WM_DRAWITEM*, trimise atunci când articolul de meniu este creat sau când i se actualizează aspectul.
MF_CHECKED	Plasează un semn de validare lângă articol. Dacă aplicația dumneavoastră posedă semne de validare de tip bitmap (vezi funcția *SetMenuItemBitmaps*), acest indicator afișează un bitmap validat lângă articolul de meniu.
MF_DISABLED	Dezactivează articolul de meniu în așa fel încât el nu mai poate fi selectat, dar nu îl umbrește.
MF_ENABLED	Activează articolul de meniu în așa fel încât el poate fi selectat și restabilește starea normală (nu mai apare umbrit).
MF_GRAYED	Dezactivează articolul de meniu și îl umbrește, în așa fel încât el nu mai poate fi selectat.
MF_MENUBARBREAK	Funcționează similar cu indicatorul *MF_MENUBREAK* pentru o bară de meniu. Pentru un meniu derulant, submeniu sau meniu cu scurtătură, noua coloană este separată de vechea coloană printr-o linie verticală.
MF_MENUBREAK	Plasează articolul pe o linie nouă (pentru barele de meniu) sau într-o coloană nouă (pentru meniurile derulante, submeniuri sau meniuri cu scurtătură) fără separarea coloanelor.
MF_OWNERDRAW	Precizează că articolul este desenat de proprietar. Înainte ca meniul să fie afișat prima oară, fereastra care deține meniul primește un mesaj *WM_MEASUREITEM* pentru a prelua lungimea și înălțimea articolului de meniu. Mesajul *WM_DRAWITEM* este atunci trimis către procedura fereastră a ferestrei proprietar, de fiecare dată când aspectul articolului de meniu trebuie actualizat.
MF_POPUP	Precizează că articolul de meniu deschide un meniu derulant sau submeniu. Parametrul *uIDNewItem* specifică identificatorul meniului derulant sau al submeniului. Acest indicator este utilizat pentru a adăuga un nume de meniu la o bară de meniu sau un articol de meniu care deschide un submeniu la un meniu derulant, submeniu sau meniu cu scurtătură.
MF_SEPARATOR	Desenează o linie orizontală de separare. Acest indicator este utilizat numai în meniuri derulante, în submeniuri sau în meniuri cu scurtătură. Linia de separare nu poate fi umbrită, dezactivată sau pusă în evidență. Windows ignoră parametrii *lpNewItem* și *uIDNewItem*

Valoare	Semnificație
MF_UNCHECKED	Nu plasează un semn de validare lângă articol (implicit). Dacă aplicația dumneavoastră posedă semne de validare de tip bitmap (vezi funcția *SetMenuItemBitmaps*), acest indicator afișează un bitmap nevalidat lângă articolul de meniu.
MF_STRING	Conține un pointer la un șir terminat în NULL.

Tabelul 1306.3 *Valorile posibile ale parametrului* ***uFlag***.

Dacă *ModifyMenu* înlocuiește un articol de meniu care deschide un meniu derulant sau un submeniu, funcția distruge vechiul meniu derulant sau submeniu și eliberează memoria utilizată de vechiul meniu. În plus, aplicația trebuie să apeleze funcția *DrawMenuBar*, de fiecare dată când se schimbă meniul, fie că meniul se află sau nu în fereastra afișată. Pentru a schimba atributele articolelor de meniu existente, este mai rapidă utilizarea funcțiilor *CheckMenuItem* și *EnableMenuItem*.

Observație: *Interfața Windows 95 API nu vă va permite să utilizați următoarele grupe de indicatoare, când invocați funcția* ***ModifyMenu***:

- ***MF_BYCOMMAND*** *și* ***MF_BYPOSITION***
- ***MF_DISABLED***, ***MF_ENABLED*** *și* ***MF_GRAYED***
- ***MF_BITMAP***, ***MF_STRING***, ***MF_OWNERDRAW*** *și* ***MF_SEPARATOR***
- ***MF_MENUBARBREAK*** *și* ***MF_MENUBREAK***
- ***MF_CHECKED*** *și* ***MF_UNCHECKED***

Pentru a înțelege mai bine procesele pe care le execută funcția *ModifyMènu*, să analizăm programul *Mod_Menu.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Programul *Mod_Menu.cpp* schimbă articolul *Test!* cu articolul *New Item!* atunci când utilizatorul îl selectează, apoi prelucrează articolul de meniu *New Item!* dacă utilizatorul a selectat articolul respectiv. De regulă, funcția *WndProc* controlează schimbarea procesului în cadrul fișierului *Mod_Menu.cpp*

Așa cum veți vedea, programul *Mod_Menu.cpp* utilizează funcția *ModifyMenu* pentru a schimba valoarea șirului articolului de meniu. În plus, codul testează identificatorii de meniu pentru a se asigura că funcția capturează articolul nou creat.

UTILIZAREA FUNCȚIEI ENABLEMENUITEM PENTRU A CONTROLA MENIURILE

C/C++1307

Așa cum ați învățat, programul dumneavoastră poate utiliza funcția *ModifyMenu* pentru controlul aspectului articolelor din meniuri. Dar utilizarea unor funcții specifice pentru abordarea unor probleme specifice duce la o mai rapidă executare a programului. Pentru a activa, a dezactiva sau a umbri articole de meniu, programul dumneavoastră poate utiliza funcția *EnableMenuItem*, mai degrabă decât funcția *ModifyMenu*. Funcția *EnableMenuItem* se utilizează cu prototipul descris mai jos:

```
BOOL EnableMenuItem(
   HMENU hMenu,          // identificator pentru meniu
   UINT uIDEnableItem, //articol meniu de activat, dezactivat
                       //sau umbrit
```

```
    UINT uEnable        //indicatoare de articol meniu
);
```

Parametrul *hMenu* este identificatorul meniului care urmează a fi modificat. Parametrul *uIDEnableItem* specifică articolul de meniu pe care îl activați, dezactivați sau umbriți, așa cum determinați prin parametrul *uEnable*. Parametrul *uIDEnableItem* specifică un articol într-o bară de meniu, într-un meniu sau submeniu. Parametrul *uEnable* specifică indicatoarele care controlează interpretarea parametrului uIDEnableItem și indică fie dacă meniul este activat, dezactivat sau umbrit. Parametrul *uEnable* trebuie să fie o combinație fie a lui *MF_BYCOMMAND*, fie *MF_BYPOSITION* și *MF_ENABLED*, *MF_DISABLED* sau *MF_GRAYED*, cum se prezintă în Tabelul 1307.

Valoare	Semnificație
MF_BYCOMMAND	Precizează că *uIDEnableItem* dă identificatorul articolului de meniu. Dacă nu este specificat nici *MF_BYCOMMAND*, nici *MF_BYPOSITION*, indicatorul *MF_BYCOMMAND* este luat ca implicit.
MF_BYPOSITION	Precizează că *uIDEnableItem* dă poziția relativă la zero a articolului de meniu.
MF_DISABLED	Precizează că articolul de meniu este dezactivat, astfel încât nu poate fi selectat, dar nu este umbrit.
MF_ENABLED	Precizează că Windows trebuie să activeze articolul de meniu astfel încât el să poată fi selectat și este restabilit din starea de umbrit.
MF_GRAYED	Precizează că articolul de meniu este dezactivat și umbrit, astfel încât el nu poate fi selectat.

Tabelul 1307 *Valorile posibile pentru parametrul* ***uEnable***.

Aplicațiile trebuie să utilizeze indicatorul *MF_BYPOSITION* pentru a specifica identificatorul corect de meniu. Dacă programul dumneavoastră specifică identificatorul de meniu la bara de meniu, Windows execută acțiunea din articolul meniului principal (adică articolul din bara de meniu). Pentru a fixa starea unui articol de meniu dintr-un meniu derulant sau submeniu prin poziție, aplicația trebuie să specifice identificatorul pentru meniul derulant sau submeniu.

Când aplicația specifică indicatorul *MF_BYCOMMAND*, Windows testează toate articolele care au deschis submeniuri în cadrul unui meniu. Atunci este suficientă specificarea identificatorului pentru meniu, în cazul în care nu există articole de meniu duplicate.

Pentru a înțelege mai bine procesele executate de funcția *EnableMenuItem*, consultați programul *Enab_Dis.cpp*, din CD-ROM-ul atașat. Programul *Enab_Dis.cpp* utilizează funcția *EnableMenuItem* pentru a activa și dezactiva articolul *IDM_ITEM1*. De regulă, funcția *WndProc* din cadrul programului *Enab_Dis.cpp* conține codul de executare a acțiunilor de activare și dezactivare, în modul descris mai jos:

```
case IDM_TEST :
   {
    HMENU hMenu = GetMenu( hWnd );
    UINT uState = GetMenuState( hMenu, IDM_ITEM1, MF_BYCOMMAND );
    if ( uState & MFS_GRAYED )
     EnableMenuItem( hMenu, IDM_ITEM1,
```

```
    MFS_ENABLED | MF_BYCOMMAND );
  else
    EnableMenuItem( hMenu, IDM_ITEM1, MFS_GRAYED | MF_BYCOMMAND );
 }
break;
```

UTILIZAREA FUNCȚIEI APPENDMENU PENTRU EXTINDEREA UNUI MENIU

C/C++1308

Aşa cum aţi învăţat, programul dumneavoastră poate executa numeroase activităţi cu meniurile, după ce programul dumneavoastră a asociat meniul cu fereastra programului. Una din cele mai frecvente acţiuni executate de programul dumneavoastră asupra unui meniu existent este să adauge articole în acel meniu. Funcţia *AppendMenu* adaugă un nou articol la capătul barei de meniu, la sfârşitul meniului derulant, submeniului sau meniului cu scurtătură specificate. Puteţi utiliza funcţia *AppendMenu*, pentru a arăta conţinutul, aspectul şi comportamentul articolului de meniu, în modul descris mai jos:

```
BOOL AppendMenu(
    HMENU hMenu,             // identificatorul meniului de modificat
    UINT uFlags,             // fanioane articole meniu
    UINT uIDNewItem,         // identificator articol meniu sau
                             // meniu derulant sau submeniu
    LPCTSTR lpszNewItem //continut articol meniu
);
```

De regulă, parametrul *hMenu* identifică o bară de meniu, un meniu derulant, submeniu sau un meniu cu scurtătură care trebuie modificate. Parametrul *uFlags* specifică indicatoarele care controlează aspectul şi comportamentul noului articol de meniu. Parametrul *uIDNewItem* precizează fie identificatorul noului articol de meniu, fie, în cazul în care parametrul *uFlags* are valoarea *MF_POPUP*, identificatorul pentru meniul derulant sau submeniu. Parametrul *lpNewItem* arată conţinutul noului articol de meniu. Interpretarea lui *lpNewItem* depinde de parametrul *uFlags* care poate să includă *MF_BITMAP*, *MF_OWNERDRAW* sau *MF_STRING*, în modul descris în Tabelul 1306.3.

Aplicaţia poate apela funcţia *DrawMenuBar* ori de câte ori se modifică un meniu, indiferent dacă meniul se află în fereastra afişată la acel moment. Programul poate stabili mai multe indicatoare pentru parametrul *uFlags*, ca în Tabelul 1308.

Valoare	Semnificaţie
MF_BITMAP	Foloseşte un bitmap ca articol de meniu. Parametrul *lpNewItem* conţine identificatorul pentru bitmap.
MF_CHECKED	Plasează un semn de validare lângă articol. Dacă aplicaţia dumneavoastră posedă semne de validare de tip bitmap, acest indicator afişează un bitmap validat lângă articolul de meniu.
MF_DISABLED	Dezactivează articolul de meniu în aşa fel încât el nu mai poate fi selectat, dar nu îl umbreşte.

Valoare	Semnificație
MF_ENABLED	Activează articolul de meniu în așa fel încât el poate fi selectat și restabilește starea normală (nu mai apare umbrit).
MF_GRAYED	Dezactivează articolul de meniu și îl umbrește, în așa fel încât el nu mai poate fi selectat.
MF_MENUBARBREAK	Funcționează similar cu *MF_MENUBREAK* pentru o bară de meniu. Pentru o un meniu derulant, submeniu sau meniu cu scurtătură, noua coloană este separată de vechea coloană printr-o linie verticală.
MF_MENUBREAK	Plasează articolul pe o linie nouă (pentru barele de meniu) sau într-o coloană nouă (pentru meniurile derulante, submeniuri sau meniuri cu scurtătură) fără separarea coloanelor.
MF_OWNERDRAW	Precizează că articolul este desenat de proprietar. Înainte ca meniul să fie afișat prima oară, fereastra care deține meniul primește un mesaj *WM_MEASUREITEM* pentru a prelua lungimea și înălțimea articolului de meniu. Mesajul *WM_DRAWITEM* este atunci trimis către procedura fereastră a ferestrei proprietar, de fiecare dată când aspectul articolului de meniu trebuie actualizat.
MF_POPUP	Precizează că articolul de meniu deschide un meniu derulant sau submeniu. Parametrul *uIDNewItem* specifică identificatorul meniului derulant sau submeniului. Acest indicator este utilizat pentru a adăuga un nume de meniu la o bară de meniu sau un articol de meniu care deschide un submeniu la un meniu derulant, submeniu sau meniu cu scurtătură.
MF_SEPARATOR	Desenează o linie orizontală de separare. Acest indicator este utilizat numai în meniuri derulante, în submeniuri sau în meniuri cu scurtătură. Linia de separare nu poate fi umbrită, dezactivată sau pusă în evidență. Funcția *AppendMenu* ignoră parametrii *lpNewItem* și *uIDNewItem* dacă specificați indicatorul *MF_SEPARATOR*.
MF_STRING	Arată că articolul de meniu este un șir de text; parametrul *lpNewItem* indică acest șir.
MF_UNCHECKED	Nu plasează un semn de validare lângă articolul (implicit). Dacă aplicația dumneavoastră posedă semene de validare de tip bitmap (vezi funcția *SetMenuItemBitmaps*), acest indicator afișează un bitmap nevalidat lângă articolul de meniu.

Tabelul 1308 *Valorile posibile ale parametrului* ***uFlags***.

Observație: *Windows nu vă va permite să utilizați următoarele grupe de indicatoare, când invocați funcția AppendMenu:*

- ***MF_BYCOMMAND*** *și* ***MF_BYPOSITION***
- ***MF_DISABLED, MF_ENABLED*** *și* ***MF_GRAYED***
- ***MF_BITMAP, MF_STRING, MF_OWNERDRAW*** *și* ***MF_SEPARATOR***
- ***MF_MENUBARBREAK*** *și* ***MF_MENUBREAK***
- ***MF_CHECKED*** *și* ***MF_UNCHECKED***

Pentru a înțelege mai bine procesele pe care le execută funcția *AppendMenu*, să analizăm programul *Add_New.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Programul *Add_New.cpp* adaugă un nou articol de meniu, *New Item*, pe propria sa linie de meniu, de fiecare dată când utilizatorul selectează articolul de meniu *Test!*. Funcția *WndProc* captează selecția *Test!* și folosește *AppendMenu* pentru a adăuga un articol nou.

```
case IDM_TEST :
  // adauga o noua optiune de meniu pe o linie noua
  AppendMenu(GetMenu(hWnd), MF_STRING |
             MF_MENUBARBREAK, 120 , "New Item");
  DrawMenuBar(hWnd);
  break;
```

UTILIZAREA FUNCȚIEI DELETEMENU PENTRU A ȘTERGE SELECȚII DIN MENIU

C/C++1309

În secțiunea 1308, ați utilizat funcția *AppendMenu* pentru a adăuga articole într-un anumit meniu. Puteți, de asemenea, utiliza în programul dumneavoastră funcția *DeleteMenu* pentru a șterge un articol dintr-un anumit meniu. Dacă articolul de meniu deschide un meniu sau un submeniu, funcția *DeleteMenu* distruge identificatorul pentru acel meniu sau submeniu și eliberează memoria utilizată de meniu sau submeniu. În programele dumneavoastră veți implementa funcția *DeleteMenu* în modul descris mai jos:

```
BOOL DeleteMenu(
HMENU hMenu,       // identificator pentru meniu
UINT uPosition,    // identificatorul sau pozitia articolului
                   // de meniu
UINT uFlags,       // indicator articol de meniu
);
```

Ca totdeauna, aplicația trebuie să apeleze funcția *DrawMenuBar* ori de câte ori modifică meniul, indiferent dacă este afișat în fereastră sau nu. Pentru a înțelege mai bine prelucrările pe care le execută comanda *DeleteMenu*, să analizăm programul *Add_Del.cpp* din CD-ROM-ul care însoțește această carte. El adaugă trei articole noi la un meniu, în momentul creării ferestrei, apoi șterge acele articole noi pe măsură ce utilizatorul le selectează. Programul controlează procesele din comutările *WM_CREATE* și *WM_COMMAND* ale funcției *WndProc*, cum se prezintă mai jos:

```
case WM_CREATE :
     {
       HMENU hMenu = GetMenu( hWnd );

       AppendMenu( hMenu, MF_STRING, IDM_ITEM1, "Item&1" );
       AppendMenu( hMenu, MF_STRING, IDM_ITEM2, "Item&2" );
       AppendMenu( hMenu, MF_STRING, IDM_ITEM3, "Item&3" );
     }
     break;

case WM_COMMAND :
```

```
switch( LOWORD( wParam ) )
  {
    case IDM_ITEM1 :
    case IDM_ITEM2 :
    case IDM_ITEM3 :
  {
   HMENU hMenu = GetMenu( hWnd );

   DeleteMenu( hMenu, LOWORD(wParam), MF_BYCOMMAND );
   DrawMenuBar( hWnd );
  }
  break;
```

1310 UTILIZAREA ARTICOLELOR DE MENIU CU TASTELE DE ACCELERARE



Programul dumneavoastră poate folosi semnul & (ampersand) în definirea unui meniu pentru a defini o tastă de accelerare pentru un articol de meniu. De asemenea, programul dumneavoastră poate adăuga acceleratori unei definiții de meniu în care apar noi articole de meniu pe măsură ce programul se desfășoară. Pentru aceasta, programul dumneavoastră va utiliza funcția *CreateAcceleratorTable*, care creează o tabelă de accelerare. Veți implementa în program funcția *CreateAcceleratorTable* conform cu următorul prototip:

```
HACCEL CreateAcceleratorTable(
     LPACCEL lpaccl,// pointer la matricea de structuri cu date
                    // acceleratori
     int cEntries   // numarul structurilor in matrice
);
```

Parametrul *lpaccl* indică o matrice de structuri *ACCEL* care descriu tabela de accelerare. Parametrul *cEntries* specifică numărul de structuri *ACCEL* în matrice. Fiecare element din structurile *ACCEL* ale matricei definește, în cadrul tabelei de accelerare, o tastă de accelerare pe care o va folosi programul. Interfața Win32 API definește structura *ACCEL* în felul următor:

```
typedef struct tagACCEL { // accl
    BYTE fVirt;
    WORD key;
    WORD cmd;
} ACCEL;
```

Tabelul 1310.1 prezintă în detaliu membrii structurii *ACCEL*. .

Membru	Descriere
FVirt	Specifică indicatoarele de accelerare. Acest membru poate fi o combinație a valorilor prezentate în Tabelul 1310.2.
Key	Specifică tasta de accelerare. Acest membru poate fi ori un cod de tastă virtuală, ori un cod de caracter ASCII.

Membru	Descriere
Cmd	Specifică identificatorul de accelerare. Această valoare este plasată în cuvântul inferior al parametrului *wParam* al mesajelor *WM_COMMAND* sau *WM_SYSCOMMAND* când se apasă tasta de accelerare.

Tabelul 1310.1 *Membrii structurii* ***ACCEL***.

Așa cum indică Tabelul 1310.1 membrul *fVirt* trebuie să fie o combinație a uneia sau mai multor valori prezentate în Tabelul 1310.2

Valoare	Semnificație
FALT	Utilizatorul trebuie să apese tasta ALT când este apăsat acceleratorul.
FCONTROL	Utilizatorul trebuie să apese tasta CTRL când este apăsat acceleratorul.
FNOINVERT	Specifică faptul că Windows nu va pune în evidență un articol de meniu principal atunci când utilizatorul utilizează acceleratorul. Dacă acest indicator nu este specificat, va fi pus în evidență, dacă se poate, articolul de meniu principal, ori de câte ori utilizatorul apasă tasta de accelerare.
FSHIFT	Utilizatorul trebuie să apese tasta Shift, când este acționată tasta de accelerare.
FVIRTKEY	Membrul *key* specifică un cod de tastă virtuală. Dacă acest indicator nu este precizat, sistemul presupune că tasta specifică un cod de caracter ASCII.

Tabelul 1310.2 *Valorile posibile ale membrului* ***fVirt***.

Pentru a înțelege mai bine procesele executate de funcția *CreateAcceleratorTable*, în secțiunea următoare veți realiza o funcție care creează o nouă tabelă de accelerare. Dacă programul dumneavoastră folosește tabele cu acceleratori, trebuie de asemenea să modificați rutinele de control al mesajelor. De exemplu, următorul cod arată o modificare simplă a buclei de mesaje care permite acum programului să accepte tabele de accelerare:

```
while (GetMessage (&msg, NULL, 0, 0))
  {
    if (!hAccel || !TranslateAcceleratorTable (hWnd, hAccel, &msg) )
    {
      TranslateMessage(&msg);
      DispatchMessage (&msg);
    }
  }
```

Variabila *hAccel* este o variabilă globală cu valoarea inițială NULL, care mai târziu va prelua identificatorul pentru tabele de accelerare. Funcția *TranslateAccelerator* translatează mesajele din tabela de acceleratori în combinația dintre *WM_COMMAND* și constantele de meniu corespunzătoare. Rețineți că programul testează funcția *TranslateAccelerator*. Dacă *TranslateAccelerator* gestionează mesajul, atunci nu bucla va fi procedura de urmat, pentru că *TranslateAccelerator* apelează funcția *WndProc* automat.

Observație: *Înainte de a încheia o aplicație, trebuie utilizată funcția* ***DestroyAcceleratorTable*** *pentru a distruge fiecare tabelă de accelerare folosită de aplicație și realizată cu funcția* ***CreateAcceleratorTable***.

1311 CREAREA UNEI TABELE DE ACCELERARE SIMPLĂ



Așa cum ați învățat în secțiunea 1310, programul dumneavoastră poate realiza tabele de accelerare pe durata execuției pentru a pune la dispoziție funcționalități suplimentare pentru meniurile produse sau modificate dinamic. Ați învățat, de asemenea, că funcția *CreateAcceleratorTable* utilizează o matrice de structuri *ACCEL* pentru a crea o tabelă de accelerare. În programul dumneavoastră, mai întâi veți crea matricea, apoi veți apela funcția *CreateAcceleratorTable*. Pentru a înțelege mai bine procesul de creare a tabelei de accelerare, să analizăm funcția *AddNewTestItem* din programul *Creat_Accel.cpp* conținut în compact discul care însoțește cartea de față, care produce o nouă intrare în tabela de accelerare de fiecare dată când utilizatorul selectează articolul *Test:*

```
#define IDM_NEWBASE 300
VOID AddNewTestItem( HWND hWnd )
{
    static int nNum = 0;
    char szMenuItem[20];
    HMENU hMenu = GetMenu( hWnd );
    ACCEL* pAccelData = NULL;
    ACCEL* pCurAccel = NULL;
    HANDLE hAccelData = NULL;
    int nNumAccel = 1;
    if ( nNum == 4 )      // Maximum 4 articole noi permise
      return;
    // Daca exista tabela de accelerare, obtin numar articole
    if ( hAccel )
      nNumAccel = CopyAcceleratorTable( hAccel, NULL, 0 ) + 1;
    // Aloca o matrice cu structuri ACCEL
    hAccelData = GlobalAlloc( GHND, sizeof(ACCEL) * nNumAccel );
    if ( hAccelData )
      pAccelData = (ACCEL*)GlobalLock( hAccelData );
    // Daca exista tabela de accelerare, copiem informatia din
    // articole in matricea recent alocata.
    if ( hAccel && pAccelData )
      {
        CopyAcceleratorTable( hAccel, pAccelData, nNumAccel-1 );
        DestroyAcceleratorTable( hAccel );
        hAccel = NULL;
      }
    // Adauga o noua optiune de meniu si o tasta de accelerare
    if ( pAccelData )
      {
        // Obtinem un pointer la noua tasta de accelerare in matrice
        pCurAccel = (ACCEL*)(pAccelData+nNumAccel-1);
```

```
            //Scriem o noua optiune de meniu
            nNum++;
            wsprintf( szMenuItem, "New Item&%d", nNum );
            AppendMenu( hMenu, MF_STRING, IDM_NEWBASE+nNum, szMenuItem );
            DrawMenuBar( hWnd );
            // Stabilim noi acceleratori F1,F2,F3 sau F4 pentru
            // noua optiune
            pCurAccel->fVirt = FNOINVERT | FVIRTKEY;
            pCurAccel->cmd = IDM_NEWBASE+nNum;
            pCurAccel->key = ( nNum == 1 ? VK_F1 :
                               nNum == 2 ? VK_F2 :
                               nNum == 3 ? VK_F3 :
                               /*default*/ VK_F4 );

            // Realizam noua tabela de accelerare
            hAccel = CreateAcceleratorTable( pAccelData, nNumAccel );
            GlobalUnlock( hAccelData );
        }
    if ( hAccelData )
      GlobalFree( hAccelData );
}
```

Funcția *AddNewTestItem* execută o mare parte a prelucrării, în mare parte explicată de comentariile din codul prezentat. Merită însă să reținem crearea elementului structurii pentru noua tastă de accelerare. În acest exemplu, nu mai mult de patru taste funcționale pot servi ca acceleratori. Limita ar putea să fie la fel de bine și 10 sau utilizatorul poate determina el tastele de accelerare.

STRUCTURA FIȘIERULUI DE RESURSE

C/C++1312

Așa cum ați învățat, teoretic toate programele Windows folosesc resurse. Pentru a organiza corect și a menține accesibilitatea acestor resurse, precum și pentru a evita încărcarea codului, veți stoca informațiile despre resursele programului într-un fișier de resurse. Fișierele de resurse sunt de asemenea eficiente și pentru că programul, de regulă, încarcă resursele în memorie numai atunci când are nevoie de aceste resurse.

Compilatorul de resurse (numit de regulă rc.exe) compilează fișierul de resurse întru-un fișier *RES*. Editorul de legături conectează fișierul *RES* la sfârșitul fișierului executabil, în întregime compilat, al programului. Toate resursele definite în fișierul de resurse devin disponibile în program pe durata execuției.

Fișierul de resurse poate include cinci tipuri de scripturi pe o singură linie: *BITMAP*, *CURSOR*, *ICON*, *POINT* și *MESSAGETABLE*. Fiecare din aceste instrucțiuni încarcă un fișier de date de tipul specificat de numele din fișierul de resurse. După ce ați inclus resursele în fișierul de resurse, programul dumneavoastră poate utiliza funcțiile *Load*, cum ar fi *LoadIcon*, pentru a avea acces la aceste obiecte. Implementarea tipică a unei resurse pe o singură linie este prezentată mai jos:

```
MYAPP ICON DISCARDABLE "GENERIC.ICO"
```

În plus față de cele cinci tipuri de scripturi resurse pe o singură linie, mai există cinci tipuri de scripturi resursă pe mai multe linii: *ACCELARATOR*, *DIALOG*, *MENU*, *RCDATA* și *STRINGTABLE*. Deja ați învățat despre tipurile *ACCELERATOR* și *MENU*. Veți învăța despre tipul *DIALOG* începând cu secțiunea 1319. Următoarele șase secțiuni arată cum veți utiliza tipurile *STRINGTABLE* și *RCDATA*.

Tipurile de fișiere resurse pe mai multe linii sunt relativ ușor de recunoscut. Fiecare tip de resursă pe mai multe linii includ o instrucțiune de tip, un bloc *BEGIN-END* și instrucțiuni în cadrul blocului *BEGIN-END* care la rândul lor pot include blocuri *BEGIN-END* suplimentare, cum prezentăm mai jos:

```
NEWMENU MENU DISCARDABLE
BEGIN
   POPUP "&File"
   BEGIN
     MENUITEM "E&xit",        IDM_EXIT
   END
   MENUITEM "&Old Menu",    IDM_OLD
END
```

1313 PREZENTAREA TABELELOR CU ȘIRURI



Așa cum ați învățat în secțiunea 1312, unul dintre tipurile multi-linie acceptat de fișierele de resurse este tipul *STRINGTABLE*. Majoritatea aplicațiilor utilizează o serie de șiruri de caractere în mesaje și în textele la ieșire. Windows pune la dispoziție tabela de șiruri ca o alternativă la metoda convențională de a plasa șiruri în zonele statice de date ale programului. Programul poate în consecință să definească șiruri de caractere în cadrul fișierului de resurse și îi poate stabili o valoare ID, ca mai jos:

```
STRINGTABLE
BEGIN
   IDS_STRING1  "Exemplu de sir simplu."
   IDS_STRING2  "Jamsa's C/C++ Programmer's Bible."
   IDS_STRING3  "Jamsa & Klander"
END
```

În plus față de definirea lor în cadrul resursei, de regulă veți defini valorile ID ale șirurilor (cum ar fi *IDS_STRING1*) într-un fișier antet separat pe care îl veți include apoi atât în fișierul resursă cât și în modulul sau modulele care vor accesa șirul. Când o aplicație trebuie să acceseze datele, veți utiliza funcția *LoadString* pentru a copia datele de tip caracter din fișierul resursă într-un buffer de memorie. Șirurile dintr-un tabel de șiruri pot conține caractere de control (cum ar fi tab, linie nouă) ca și caractere de imprimantă.

Există un număr apreciabil de avantaje pentru programator în utilizarea acestor tabele. Principalul avantaj al utilizării tabelelor de șiruri este reducerea utilizării memoriei în program. Deoarece programul dumneavoastră nu încarcă șirurile decât în momentul când are nevoie de ele, nu este necesar să stocăm șirurile în zonele de date statice ale programului. Din acest motiv, trebuie să evitați să copiați tabela cu șirurile de date în bufferul de memorie statică pentru că procedând astfel, veți afecta scopul tabelei de șiruri. Puteți în

schimb să copiați datele de tip șir din tabele de șiruri în variabile locale (cum ar fi o variabilă de stivă) sau în memoria alocată global.

Un alt avantaj semnificativ al utilizării tabelelor de șiruri este sprijinul pe care ele îl oferă utilizării mai multor limbi. Interfața Win32 API acceptă resurse pentru mai multe limbi într-o singură aplicație, putând distribui același program executabil în mai multe țări, fără să îl schimbați. Secțiunile următoare vor utiliza tabele de șiruri pentru a păstra informațiile despre ferestre.

RESURSELE PERSONALIZATE

C/C++1314

Așa cum ați învățat în secțiunea 1312, unul dintre tipurile de resurse multi-linie este tipul *RCDATA*. Puteți utiliza tipul *RCDATA* pentru a stoca alte tipuri de date statice, în special date binare. De exemplu, următorul cod stochează mai multe fragmente de date binare de diferite tipuri, în resursa *DataID*.

```
DataID RCDATA
BEGIN
   3
   40
   0X8232
   "Sir de date (continuare)..."
   "Alte siruri de date\0"
END
```

Deoarece puteți include resurse de date personalizate (*custom*) direct în fișierul programului sau să le citiți dintr-un fișier de date extern, cel mai bun loc pentru a stoca resurse de date personalizate este un fișier extern. Compilatorul, în timp ce compilează fișierul de resurse, poate să adauge conținutul fișierului extern la resursele de date. De exemplu, următoarele două instrucțiuni definesc tipurile *TEXT* și *METAFILE*, atribuie câte un identificator fiecărui tip, apoi adaugă resursa fișierului de resurse.

```
happy    TEXT       "happydog.txt"
picture  METAFILE   "happypic.wmf"
```

Când definiți tipuri de resurse personalizate veți utiliza funcția *FindResource* împreună cu funcția *LoadResource* pentru a încărca resursele personalizate în program.

ÎNCĂRCAREA ÎN PROGRAM A TABELELOR DE ȘIRURI CU FUNCȚIA LOADSTRING

C/C++1315

Așa cum ați învățat, puteți crea tabele cu șiruri în fișierele de resurse. Veți atribui fiecărui șir din tabelă o valoare ID specifică, pe care de regulă o veți defini în cadrul fișierului antet al programului, în modul prezentat mai jos:

```
#define IDM_EXIT     100
#define IDM_TEST 200
#define IDM_ABOUT    301
```

După ce aţi definit o tabelă de şiruri, programul dumneavoastră va apela funcţia *LoadString*. Ea încărcă o resursă de tip şir din fişierul executabil asociat cu modulul respectiv, copiază şirul într-un buffer şi adaugă bufferului caracterul de terminare NULL. Funcţia *Loadstring* va fi utilizată în programe în forma prezentată mai jos:

```
int LoadString(
HINSTANCE hInstance, // identificator pentru modulul cu resursa
UINT uID,            // identificator resursa
LPTSTR lpBuffer,     // adresa buffer resursa
int nBufferMax       // dimensiune buffer
);
```

Parametrul *hInstance* identifică instanţa modulului al cărui fişier executabil conţine resursa şir. Parametrul *uID* specifică un identificator întreg al şirului care se încarcă. Parametrul *lpBuffer* indică bufferul care primeşte şirul. În sfârşit, parametrul *nBufferMax* specifică dimensiunea bufferului în octeţi (versiunea ANSI) sau în caractere (versiunea UNICODE). Funcţia trunchiază şirul şi îl încheie cu NULL dacă este mai lung decât numărul de caractere specificat.

Dacă funcţia *LoadString* reuşeşte, valoarea returnată va fi numărul de octeţi (versiunea ANSI) sau numărul de caractere (versiunea UNICODE) copiate în buffer, fără includerea caracterului terminator (NULL)sau zero, dacă resursele de tip şir nu există. Pentru informaţii mai extinse despre erori, apelaţi funcţia *GetLastError*. Pentru a înţelege mai bine funcţia *LoadString* să analizăm următorul fragment de cod din programul *LoadStrg.cpp* din CD-ROM-ul ataşat la cartea de faţă.

```
case IDM_TEST :
     {
        char szString[40];
        SHORT idx;
        for ( idx = IDS_STRINGBASE; idx < IDS_STRINGBASE+3;idx++ )
          {
            LoadString( hInst, idx, szString, 40 );
            MessageBox( hWnd, szString, "String Loaded", MB_OK |
               MB_ICONINFORMATION );
          }
     }
 break;
```

Programul *LoadStrg.cpp* va încărca şi afişa trei şiruri separate atunci când utilizatorul va selecta opţiunea *Test*. Programul va afişa fiecare şir în cadrul unei casete de mesaj.

1316 LISTAREA CONŢINUTULUI UNUI FIŞIER RESURSĂ

C/C++

Aşa cum aţi învăţat, în fişierele resursă puteţi stoca mai multe tipuri diferite de date personalizate. Funcţia *EnumResourceNames* caută în modul fiecare resursă specificată în parametrul *lpszType* şi transmite numele fiecărei resurse regăsite la o funcţie *callback* definită în aplicaţie. Funcţia *EnumResourceName* continuă să enumere până când funcţia

callback returnează *false* sau până când *EnumResourceNames* a întâlnit toate numele de resurse. În programele dumneavoastră, veți utiliza funcția *EnumResourceNames* în forma prezentată mai jos:

```
BOOL EnumResourceNames(
     HINSTANCE hModule, // identificator modul resursa
     LPCTSTR lpszType,  // pointer la un tip de resursa
     ENUMRESNAMEPROC lpEnumFunc, // pointer la o functie
                                 // callback
     LONG lParam        // parametru definit in aplicatie
);
```

Funcția *EnumResourceNames* acceptă parametrii prezentați în Tabelul 1316.1.

Parametru	Descriere
hModule	Identifică modulul al cărui fișier executabil conține resursele pe care trebuie să le enumere *EnumResourceNames*. Dacă acest parametru este NULL, funcția enumeră numele resurselor din modul folosite la crearea procesului curent.
lpszType	Indică un șir terminat în NULL care precizează numele tipului resursei pentru care numele este enumerat. Pentru tipurile de resurse standard, parametrul ia una din valorile prezentate în Tabelul 1316.2.
lpEnumFunc	Indică o funcție *callback* care va fi apelată pentru fiecare nume de resursă întâlnit.
lParam	Specifică o valoare definită de aplicație transmisă funcției callback. Programul dumneavoastră poate utiliza acest parametru când sunt erori de testare.

Tabelul 1316.1 *Parametrii funcției* ***EnumResourceNames***.

Așa cum menționează Tabelul 1316.1 parametrul *lpszType* poate avea una din valorile prezentate în Tabelul 1316.2.

Valoare	Semnificație
RT_ACCELERATOR	Tabelă de accelerare
RT_ANICURSOR	Cursor animat
RT_ANIICON	Pictogramă animată
RT_BITMAP	Resursă bitmap
RT_CURSOR	Resursă tip cursor dependentă de hardware
RT_DIALOG	Casetă de dialog
RT_FONT	Resursă tip font
RT_FONTDIR	Resursă cu directorul de fonturi
RT_GROUP_CURSOR	Resursă tip cursor independentă de hardware
RT_GROUP_ICON	Resursă tip pictogramă independentă de hardware
RT_ICON	Resursă tip pictogramă dependentă de hardware
RT_MENU	Resursă tip meniu
RT_MESSAGETABLE	Intrare în tabelă de mesaje

(continuare)

Valoare	Semnificație
RT_PLUGPLAY	Resursă tip plug-and-play
RT_RCDATA	Resursă definită de aplicație (date de rând)
RT_STRING	Intrare în tabelă de șiruri
RT_VERSION	Resursă tip versiune
RT_VXD	VXD (Virtual Device Driver)

Tabela 1316.2 *Valorile posibile ale parametrului* ***lpszType***.

Pentru a înțelege mai bine cum operează funcția *EnumResourceNames*, să analizăm programul *3_pics.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Fișierul resursă pentru programul *3_pics.cpp* cuprinde trei elemente bitmap. Programul utilizează funcția *EnumResourceNames* și o funcție callback numită *PaintBitmaps* pentru a enumera resursele de tip *RT_BITMAP* din fișierul resursă *3_pics.rc*. Următorul fragment de cod prezintă funcția *PaintBitmaps:*

```
BOOL CALLBACK PaintBitmaps(HANDLE hModule, LPCTSTR lpszType,
                           LPCTSTR lpszName, LONG lParam )
  {
    HBITMAP hBitmap = LoadBitmap( hModule, lpszName );
    if ( hBitmap )
      {
       BITMAP bm;
       HDC hMemDC;
       HWND hWnd = (HWND)lParam;
       HDC hDC = GetDC( hWnd );
       HFONT hOldFont;
       // Obtine dimensiunea pentru bitmap.
       GetObject( hBitmap, sizeof( BITMAP ), &bm );
       // Produce un DC pt. selectare bitmap.
       hMemDC = CreateCompatibleDC( hDC );
       SelectObject( hMemDC, hBitmap );
       // Afisaza bitmap, extins la 50X50 pixeli.
       StretchBlt( hDC, gnPos, 0, 50, 50,
              hMemDC, 0, 0, bm.bmWidth, bm.bmHeight,
              SRCCOPY );
       // Afisaza nume bitmap.
       hOldFont = SelectObject( hDC,
         GetStockObject( ANSI_VAR_FONT ) );
       TextOut( hDC, gnPos, 60, lpszName, strlen( lpszName ) );
       SelectObject( hDC, hOldFont );
       DeleteDC( hMemDC );
       ReleaseDC( hWnd, hDC );
       DeleteObject( hBitmap );
       gnPos += 100;
      }
    return( TRUE );
  }
```

Funcția callback *PrintBitmaps* execută prelucrări importante. Multe din ele vor fi explicate în detaliu în următoarele secțiuni. Funcția pictează fiecare bitmap pe care funcția *EnumResourceTypes* îl enumeră pe ecran.

UTILIZAREA FUNCȚIEI ENUMRESOURCETYPES CU FIȘIERELE DE RESURSE

C/C++1317

În secțiunea 1316 ați învățat despre funcția *EnumResourceNames* utilizată de program pentru a enumera diferite resurse de un anumit tip dintr-un fișier de resurse. Vor fi însă ocazii când programul dumneavoastră nu va ști anticipat ce resurse conține fișierul de resurse. În aceste situații, programul poate utiliza funcția *EnumResourceTypes*. Această funcție caută resursele în modul și transmite fiecare tip de resursă găsită către o funcție callback definită de aplicație. Funcția *EnumResourceTypes* continuă să enumere tipuri de resurse până când funcția callback returnează *false* sau până a enumerat toate tipurile de resurse. Implementarea funcției *EnumResourceTypes* se face în modul prezentat mai jos:

```
BOOL EnumResourceTypes(
    HMODULE hModule,  // identificator de modul resursa
    ENUMRESTYPEPROC lpEnumFunc,  // pointer la o functie
                                 // callback
    LONG lParam          // parametru definit de aplicatie
);
```

Așa cum funcția *EnumResourceNames* utilizează o funcție callback, la fel procedează și funcția *EnumResourceTypes*. Prototipul funcției callback pe care o veți folosi cu funcția *EnumResourceTypes* este prezentat mai jos (numele funcției *EnumResTypesProc* este un înlocuitor pentru numele funcției definite în aplicație sau pentru cea definită în bibliotecă).

```
BOOL CALLBACK EnumResTypeProc(
    HANDLE hModule,  // identificator de modul resursa
    LPTSTR lpszType, // pointer la un tip de resursa
    LONG lParam       // parametru definit de aplicatie
);
```

Parametrul *hModule* identifică modulul al cărui fișier executabil conține resursele al căror tip trebuie enumerat. Dacă parametrul e NULL, funcția enumeră tipurile de resurse din modulul utilizat pentru crearea procesului curent. Parametrul *lpszType* indică un șir de caractere terminat în NULL care specifică numele de tip al resursei ce va fi enumerată. Pentru tipurile de resurse standard, acest parametru poate lua una din valorile prezentate în Tabelul 1316.2.

Pentru a înțelege mai bine modul cum operează funcția *EnumResourceTypes*, să analizăm programul *EnumResT.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Acest program completează o listă cu toate tipurile de resurse pe care le enumeră în fișierul de resurse dat. Următorul fragment de cod detaliază funcția callback *ListResourceTypes*, pe care funcția *EnumResourceTypes* o utilizează ca funcție callback:

```
BOOL CALLBACK ListResourceTypes( HANDLE hModule, LPTSTR
   lpszType, LONG lParam )
{
   LPTSTR lpAddString = lpszType;
```

```
    HWND hListBox = (HWND)lParam;
    // verifica daca tipul resursei este un tip predefinit. Daca da,
    // stabileste lpAddString la un sir descriptiv.
    switch( LOWORD(lpszType) )
      {
        case RT_ACCELERATOR : lpAddString = "Accelerator"; break;
        case RT_BITMAP      : lpAddString = "Bitmap"; break;
        case RT_DIALOG      : lpAddString = "Dialog"; break;
        case RT_FONT        : lpAddString = "Font"; break;
        case RT_FONTDIR     : lpAddString = "FontDir"; break;
        case RT_MENU        : lpAddString = "Menu"; break;
        case RT_RCDATA      : lpAddString = "RC Data"; break;
        case RT_STRING      : lpAddString = "String Table"; break;
        case RT_MESSAGETABLE : lpAddString = "Message Table"; break;
        case RT_CURSOR      : lpAddString = "Cursor"; break;
        case RT_GROUP_CURSOR : lpAddString = "Group Cursor"; break;
        case RT_ICON        : lpAddString = "Icon"; break;
        case RT_GROUP_ICON  : lpAddString = "Group Icon"; break;
        case RT_VERSION     : lpAddString = "Version Information";
                              break;
      }
    SendMessage( hListBox, LB_INSERTSTRING, (WPARAM)-1,
                 (LPARAM)lpAddString );
    return( TRUE );
  {
```

Funcția callback primește valoarea *lpszType* de la funcția *EnumResourceTypes* și convertește valoarea la o valoare șir de caractere, cum ar fi „Icon" sau „Menu", mai ușor de identificat de utilizator.

1318 ÎNCĂRCAREA RESURSELOR ÎN PROGRAM CU FINDRESOURCE

C/C++

Așa cum ați învățat, programul dumneavoastră poate defini orice resursă personalizată în cadrul fișierului de resurse. În secțiunile precedente ați utilizat funcțiile *EnumResourceNames* și *EnumResourceTypes* pentru a lista toate numele de resurse ale unui tip dat și toate tipurile de resurse din cadrul unui fișier de resurse dat. Opțional, programul poate utiliza funcțiile *FindResource* și *LoadResource* pentru a îndeplini aceeași sarcină. Funcția *FindResource* determină locația unei resurse al cărei nume și tip îl specificați în modulul respectiv. Funcția *FindResource* va fi utilizată în programul dumneavoastră cu prototipul prezentat mai jos:

```
HRSRC FindResource(
HMODULE hModule,  // identificator de modul resursa
LPCTSTR lpName,   // pointer la un nume de resursa
LPCTSTR lpType    // pointer la un tip de resursa. Vezi
                  // Tabel 1316.2
);
```

Dacă funcția *FindResource* reușește, valoarea returnată este un identificator la blocul de informații al resursei. Pentru a obține un identificator al resursei, transmiteți identificatorul returnat de *FindResource,* funcției *LoadResource.* Dacă funcția eșuează, valoarea returnată va fi *NULL.*

Dacă în parametrii *lpName* sau *lpType,* cuvântul cel mai semnificativ este zero, atunci cuvântul mai puțin semnificativ va specifica identificatorul întreg al numelui sau tipului de resursă dată. În caz contrar, acești parametri vor fi pointeri de tip *long* la un șir de caractere terminat cu *NULL.* Dacă primul caracter al șirului este semnul #, restul caracterelor va reprezenta un număr zecimal care specifică identificatorul de tip întreg al numelui sau tipului resursei. De exemplu, șirul "#258" reprezintă identificatorul întreg 258.

Aplicația dumneavoastră trebuie să reducă volumul de memorie cerut de resursă prin referirea la identificatorul de tip întreg al resursei și nu la numele ei. Aplicațiile pot utiliza funcția *FindResource* pentru a găsi orice tip de resursă. E recomandabil să utilizați *FindResource* numai dacă trebuie să accesați resursa sub formă de date binare, în cazul unor apeluri succesive la funcțiile *LoadLibrary* și *LoadResource.* Pentru a învăța mai mult despre *LoadLibrary* și *LoadResource* consultați documentația on-line a compilatorului.

Pentru a utiliza resursa imediat, aplicațiile trebuie să folosească una din funcțiile specifice resurselor. Tabelul 1318 prezintă cum se caută și încarcă resursa într-un singur apel:

Funcție	Acțiune
FormatMessage	Încarcă și formatează intrarea unei tabele de mesaje
LoadAccelerators	Încarcă o tabelă de accelerare
LoadBitmap	Încarcă o resursă bitmap
LoadCursor	Încarcă o resursă de tip cursor
LoadIcon	Încarcă o resursă de tip pictogramă
LoadMenu	Încarcă o resursă de tip meniu
LoadString	Încarcă o resursă de tip șir de caractere

Tabelul 1318 *Funcțiile specifice de încărcare a resurselor.*

De exemplu, o aplicație poate utiliza funcția *LoadIcon* pentru a încărca o pictogramă și a o afișa pe ecran. Aplicația va trebui însă să utilizeze *FindResource* și *LoadResource* dacă încarcă pictograma pentru a copia datele într-o altă aplicație.

Pentru a înțelege mai bine cum operează funcția *FindResource,* să analizăm programul *FindRes.cpp,* conținut în CD-ROM-ul care însoțește cartea de față. *FindRes.cpp* utilizează *FindResource* pentru a încărca un rând de date binare într-o structură. Programul *FindRes.cpp* prezintă modificări în fișierul de resurse, antet și de program, față de fișierele inițiale din proiectul *generic.* Fișierul resursă include următorul cod suplimentar, care definește într-un mod simplu un grup de valori hexazecimale:

```
TestData RCDATA DISCARDABLE
BEGIN
  0x0001
  0x0002
  0x0003
END
```

Fişierul antet defineşte o structură de date în care programul va citi datele din blocul *TestData*, ca mai jos:

```
typedef struct
{
   SHORT Value1;
   SHORT Value2;
   SHORT Value3;
}RESDATA;
```

În final, fişierul program *FindRes.cpp* citeşte datele în structură, în cadrul codului de tratare a articolului de meniu *Test!* din funcţia *WndProc*, ca mai jos:

```
case IDM_TEST :
       {
          HRSRC hres = FindResource( hInst, "TestData",
           RT_RCDATA );
          if ( hres )
           {
           char szMsg[50];
           DWORD size = SizeofResource( hInst, hres );
           HGLOBAL hmem = LoadResource( hInst, hres );
           RESDATA* pmem = (RESDATA*)LockResource( hmem );
           wsprintf( szMsg, "Values loaded: %d, %d,
             %d\nSize = %d",
                pmem->Value1, pmem->Value2, pmem->Value3,
                  size );
           MessageBox( hWnd, szMsg, lpszAppName, MB_OK );
          }
       }
   break;
```

Dacă programul găseşte datele despre resursă sub cuvântul cheie *Test!Data*, va determina volumul datelor şi le va salva în variabila *size*. Programul va utiliza atunci specificatorul *HGLOBAL* pentru a aloca spaţiu în memoria *heap* pentru stocarea datelor. În sfârşit, el va stoca datele într-o instanţă a structurii *RESDATA*. După compilarea şi executarea programului, în cazul selectării articolului de meniu *Test!*, rezultatul va apărea pe ecran într-o casetă de mesaj.

1319 CASETELE DE DIALOG

C/C++

De-a lungul secţiunilor precedente aţi produs diferite ferestre în cadrul programelor dumneavoastră. Dar majoritatea ferestrelor create au fost ferestre principale, al căror cod îl doreaţi să opereze în cadrul execuţiei programului. O fereastră principală tratează mesajele de diverse tipuri pe parcursul executării programului.

O casetă de dialog, pe de altă parte este similară cu o fereastră *pop-up*. O casetă de dialog preia intrările de la utilizator pentru o anumită sarcină, cum ar fi obţinerea unui fişier sau a unui şir de caractere pentru o căutare. Cea mai importantă diferenţă dintre casetele de dialog şi ferestrele *pop-up*, constă în aceea că o casetă de dialog utilizează *şabloane* care definesc

controalele afișate de casetă. Veți folosi tipul de resursă *DIALOG* sau veți defini aceste șabloane în cadrul fișierului de resurse. Veți putea crea, de asemenea, șabloane în mod dinamic în memorie la executarea programului.

Casetele de dialog operează diferit de ferestrele *pop-p* și pentru că aceste casete folosesc o funcție implicită de prelucrare a mesajelor care interpretează evenimentele de la tastatură cum ar fi apăsarea tastelor cu săgeată sau TAB, facilitând selectarea elementelor de control din casetă de către utilizator. De regulă, funcția suplimentară de prelucrare a mesajelor și aspectul casetei de dialog fac ca aceasta să fie mai convenabilă pentru acceptarea și prelucrarea intrărilor de la utilizator. În următoarele secțiuni veți învăța despre tipurile de casete de dialog și modalitățile lor de utilizare în programe.

DEFINIREA TIPURILOR DE CASETE DE DIALOG

C/C++ 1320

Așa cum ați aflat din secțiunea 1319, puteți utiliza casete de dialog în cadrul programelor dumneavoastră pentru a oferi informații către utilizator și a primi în schimb informații de la acesta. Există două tipuri de casete de dialog: casete de dialog *modale* și casete de dialog *nemodale*.

Când este afișată pe ecran o casetă de dialog modală, utilizatorul nu poate comuta la o altă secțiune a programului până când nu închide caseta de dialog. În mod implicit, caseta de dialog limitează accesul la alte ferestre vizibile ale aplicației care a apelat caseta. Utilizatorii însă, pot comuta la alte aplicații, dacă o singură aplicație afișează caseta de dialog modală. Cea mai simplă casetă de dialog este caseta de mesaje utilizată în secțiunile anterioare.

Programul dumneavoastră poate specifica și casete de dialog *modale de sistem*. Caseta de dialog modală de sistem preia controlul întregului ecran și nu permite utilizatorului să execute procese suplimentare din orice alt program, până când utilizatorul nu răspunde la acea casetă de dialog. Utilizarea casetelor de dialog modale de sistem în cadrul programului dumneavoastră se impune numai în cazuri severe ce nu pot fi ignorate de utilizator, cum ar fi erorile de sistem. Pentru a crea casete de dialog modale de sistem, puteți specifica fie stilul *WS_SYSMODAL* în caseta de dialog șablon, fie să creați caseta cu ajutorul funcției *SetSysModalWindow*.

De obicei însă, pentru a crea casete de dialog modale și modale de sistem, veți utiliza funcția *DialogBox*. Când programul dumneavoastră utilizează această funcție, Windows trimite toate mesajele ferestrei apelante la caseta de dialog.

Mai puțin frecvente, dar încă utilizate, sunt casetele de dialog *nemodale*. Spre deosebire de caseta modală, caseta de dialog nemodală poate primi sau pierde intrările de la utilizator (adică poate fi activă sau poate fi dezactivată). Caseta nemodală are o durată de viață nedefinită. Deoarece însă, caseta de dialog nemodală poate exista pe o perioadă îndelungată de timp, programele care utilizează astfel de casete trebuie să se asigure că bucla de mesaje partajează mesajele cu casetele de dialog nemodale. Modul în care veți construi bucla de mesaje pentru programele care conțin casete nemodale, este prezentat mai jos:

```
while( GetMessage( &msg, NULL, 0, 0) )
  if (hDlgModeless || !IsDialogMessage(hDlgModeless, &msg))
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
```

În construcția generală, valoarea *hDlgModeless* reprezintă un identificator pentru caseta de dialog nemodală. Dacă această casetă nu este deschisă în acel moment, valoarea *hDlgModaless* trebuie să fie *NULL*. Funcția *IsDialogMessage* determină dacă vreun mesaj din Windows este destinat casetei de dialog. Dacă da, funcția va trimite mesajul către procedura de prelucrare a mesajului proprie casetei de dialog și, în consecință, funcțiile *TranslateMessage* și *DispatchMessage* nu vor mai prelucra mesajul.

1321 UTILIZAREA TASTATURII CU CASETELE DE DIALOG

C/C++

Așa cum ați învățat, Windows dispune de un mecanism logic intern de prelucrare a casetelor de dialog în forma procedurii speciale de prelucrare a mesajului casetei de dialog. Mecanismul logic include și mijloace de selectare a elementelor din caseta de dialog, prin tastatură în loc de mouse. Programele dumneavoastră pot pune la dispoziție trei modalități de acționare a tastaturii: prin taste marcate (*hot keys*) pentru selectarea articolelor prin combinația rezultată din tasta Alt+tastă de literă, prin tasta TAB pentru navigare între elementele de control și prin tastele cu săgeți pentru navigare între elementele de control și în cadrul acestora.

În cadrul casetelor de dialog, veți oferi suport pentru combinația Alt+tastă de literă, în aceeași modalitate ca la articolele de meniu. În șirul de text al controlului, puneți semnul & în fața literei pe care o doriți în combinație cu tasta Alt. De exemplu, definiția controlului *DEFPUSHBUTTON* va utiliza drept combinație Alt+D pentru a activa butonul *Dalmatian*:

```
CONTROL "&Dalmatian", IDC_DONE, "BUTTON",
     BS_DEFPUSHBUTTON | WS_TABSTOP | WS_CHILD, 45, 66, 48, 12
```

Utilizatorii pot, în anumite momente, considera mai convenabilă tastatura decât mouse-ul. Însă, la fel ca multe alte funcții simplificate de utilizator, nu definiți prea multe scurtături de la tastatură (*keyboard shortcuts*) pentru ele că pot crea confuzii în loc să ajute. Pentru a accepta controlul de la tastatură în cadrul casetei de dialog, trebuie să definiți anumite elemente ale casetei șablon, incluzând stilurile *WS_TABSTOP* și *WS_GROUP*.

Stilul *WS_TABSTOP* marchează fiecare element care va deveni activ când utilizatorul apasă pe TAB sau pe SHIFT+TAB. Stilul *WS_GROUP* marchează începutul unui grup. Toate elementele listate în scriptul de resurse până la următorul stil *WS_GROUP* sunt parte a unui singur grup. Utilizatorul poate folosi tastele cu săgeți pentru a naviga printre elementele grupului, dar nu poate folosi tastele cu săgeți pentru deplasarea de la un grup la altul.

1322 COMPONENTELE ȘABLONULUI DE CASETĂ DE DIALOG

C/C++

Așa cum ați învățat, în cadrul unui fișier de resurse veți scrie instrucțiunea *DIALOG* în fața unei serii de elemente de control ale casetei de dialog pentru a defini caseta de dialog. Forma generală a casetei de dialog șablon este similară cu resursele șablon pe care le-ați utilizat în secțiunile precedente pentru a proiecta meniuri și articole de meniu. Forma generală a casetei de dialog șablon este prezentată mai jos:

```
IdentificatorCD DIALOG DISCARDABLE stanga, sus, lungime, inaltime
STYLE Stil1 | Stil2 | . . . | StilN
```

```
CAPTION "Titlul dialogului"
FONT Dimensiune Font, "Nume Font"
BEGIN
TipControl1 "TextControl", ID_Control , stanga, sus, lungime,
  inaltime
TipControl2 "TextControl", ID_Control , stanga, sus, lungime,
  inaltime
. . .
TipControl3 "TextControl", ID_Control , stanga, sus, lungime,
  inaltime
END
```

Tipul de control *TipControl* se referă la unul din mai multe tipuri de ferestre copil folosite pentru a crea elemente de control în casetele de dialog. Intrarea *TipControl* trebuie să conțină una dintre valorile listate în Tabelul 1322.

Tipuri de control acceptate

BUTTON	*CHECKBOX*	*COMBOBOX*	*CONTROL*
CTEXT	*DEFPUSHBUTTON*	*EDITTEXT*	*GROUPBOX*
ICON	*LISTBOX*	*LTEXT*	*PUSHBUTTON*
RADIOBUTTON	*RTEXT*	*SCROLLBAR*	*STATIC*

Tabelul 1322 *Valorile acceptate ale intrării* ***TipControl*** *din definiția casetei de dialog.*

CREAREA UNUI ȘABLON PENTRU CASETE DE DIALOG

C/C++1323

În secțiunea 1322 ați învățat forma generică a declarației casetei de dialog care include multe componente importante. Analiza unei declarații pentru o casetă de dialog simplă vă poate ajuta în a înțelege definirea casetei de dialog, ca mai jos:

```
TESTDIALOG DIALOG DISCARDABLE 20, 20, 180, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
  WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
BEGIN
  CHECKBOX     "Check box control.",IDC_CHECKBOX,9,7,70,10
  GROUPBOX     "Radio Buttons",-1,7,21,86,39
  RADIOBUTTON  "First", IDC_RADIO1,13,32,37,10,WS_GROUP |
                WS_TABSTOP
  RADIOBUTTON  "Second",IDC_RADIO2,13,45,39,10
  PUSHBUTTON   "Done",IDCANCEL,116,8,50,14,WS_GROUP
END
```

Declarația precedentă produce caseta de dialog prezentată în Figura 1323.

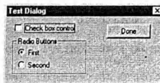


Figura 1323 *Caseta de dialog* ***Test Dialog***.

Dacă vă uitați mai atent la caseta de dialog rezultată, veți constata că înțelegeți mult mai ușor componentele definite în fișierul de resurse. Fiecare instrucțiune din blocul *BEGIN-END* produce un control nou. Prima instrucțiune produce caseta de validare în partea stângă de sus. De asemenea, alocă la acel control identificatorul *IDC_CHECKBOX*, astfel încât tratarea mesajelor poate răspunde adecvat la modificările intervenite în valoarea controlului (în cazul nostru, dacă respectivul control este marcat cu semnul de validare sau nu).

A doua instrucțiune creează chenarul butoanelor radio, imposibil de selectat de către utilizator, astfel încât definiția nu îi alocă un identificator. A treia și a patra instrucțiune produc butoanele radio *First* și *Second* în cadrul chenarului. Aceste butoane sunt într-un grup; utilizatorul poate selecta numai un singur buton la un moment dat. Observați cum grupul începe cu butonul *IDC_RADIO1*, ca apoi fișierul de resurse să declare un alt grup cu butonul *Done*.

Ultima instrucțiune din bloc utilizează identificatorul *IDCANCEL* la crearea butonului *Done*. Ca regulă, veți aloca identificatorul *IDCANCEL* fiecărui buton care închide caseta de dialog fără să salveze schimbările intervenite.

1324 COMPONENTELE DEFINIȚIEI CASETEI DE DIALOG



Așa cum ați sînvățat în secțiunea 1323, orice definiție de casetă de dialog operează prelucrări specifice în blocul *BEGIN-END*, pentru a crea controalele pe care le va afișa caseta de dialog. Înainte de a proceda la definirea controlului, caseta de dialog își definește propriile atribute standard. De exemplu, caseta de dialog *TESTDIALOG* din secțiunea 1323 începe cu aceste prime patru definiții:

```
TESTDIALOG DIALOG DISCARDABLE 20, 20, 180, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
  WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
```

Prima linie identifică o casetă de dialog cu identificatorul *TESTDIALOG*. Aceeași linie declară dimensiunile casetei de dialog. În exemplul anterior, caseta de dialog începe la 20 DBU (*dialog base units* - unități de casetă de dialog) în jos și 20 DBU de la extremitatea interioară a ferestrei client apelante. Caseta de dialog are, de asemenea, 180 DBU în lățime și 70 DBU în adâncime.

A doua linie definește stilurile utilizate în caseta de dialog în momentul realizării ei. Puteți utiliza stilurile definite în secțiunea 1272, care încep fie cu *WS (window style)*, fie cu *DS*

(dialog style). Întotdeauna trebuie să includeţi stilul *WS_VISIBLE* pentru a face vizibil dialogul. Nu puteţi aplica stilurile *WS_MINIMIZEBOX* şi *WS_MAXIMIZEBOX* în casetele de dialog.

Veţi utiliza specificatoarele de titlu prezentate în linia a treia (*CAPTION*) numai la casetele de dialog declarate cu stilul *WS_CAPTION*. De regulă, două raţiuni motivează necesitatea titlurilor în casetele de dialog. Prima, titlul face cunoscut utilizatorului scopul casetei de dialog. A doua, permite utilizatorului să deplaseze caseta de dialog pe ecran. Pentru a simplifica interacţiunea utilizatorului cu programul dumneavoastră, nu uitaţi să puneţi între ghilimele şirul titlu.

În sfârşit, specificatorul *FONT* determină nu numai tipul de font utilizat de caseta de dialog, ci şi dimensionarea fiecărui control din casetă, precum şi a casetei înseşi. Specificatorul de font joacă un rol atât de important în dimensionare, datorită faptului că Windows calculează unitatea de bază a casetei de dialog (DBU) ca fracţiune a dimensiunii fontului. Pentru majoritatea casetelor de dialog, o bună alegere ar fi fontul MS Sans Serif de 8 puncte. Trebuie să vă asiguraţi că numele fontului dintre ghilimele corespunde exact numelui de font definit de sistem, altfel fişierul de resurse nu va fi corect compilat.

DEFINIREA CONTROALELOR DIN CASETELE DE DIALOG

C/C++1325

Sistemul Windows vă pune la dispoziţie două modalităţi de definire a controalelor din casetele de dialog. Prima modalitate este să utilizaţi o instrucţiune explicită, cum ar fi instrucţiunea *COMBOBOX*, cum am văzut în secţiunea 1323. Cealaltă modalitate este utilizarea instrucţiunii *CONTROL* şi includerea stilului de casetă combinată ca parametru. De exemplu, următoarele două instrucţiuni produc acelaşi control:

```
CONTROL "Apasa aici", IDC_BUTTON1, "button",
    BS_DEFPUSHBUTTON | WS_TABSTOP | WS_CHILD, 45, 66, 48, 12
DEFPUSHBUTTON "Apasa aici", IDC_BUTTON1, 45, 66, 48, 12,
    WS_TABSTOP
```

Fiecare dintre definiţii este acceptabilă, însă, aşa cum aţi învăţat în multe secţiuni precedente, trebuie să vă decideţi asupra unui stil specific, pe care îl veţi utiliza pe întreg parcursul fişierului de resurse.

În mod normal, veţi defini valorile ID pentru fiecare control într-un fişier antet separat. Pentru controalele cu parametrul opţional *style*, opţiunile includ stilurile *WS_TABSTOP* şi *WS_GROUP*. Stilurile *WS_TABSTOP* şi *WS_GROUP* controlează interfaţa implicită cu tastatura, descrisă în secţiunea 1321. Trebuie să folosiţi operatorul SAU (OR) pe bit (|) pentru a combina toate stilurile pe care le alocaţi unui control.

UTILIZAREA MACROCOMENZII DIALOGBOX PENTRU A AFIŞA O CASETĂ DE DIALOG

C/C++1326

Aşa cum aţi învăţat, programul dumneavoastră va defini şabloane de casete de dialog în fişierele de resurse. În următoarele câteva secţiuni, veţi învăţa despre alte modalităţi de afişare a casetelor de dialog. Dintre cele mai simple metode pe care le poate utiliza programul dumneavoastră este macrocomanda *DialogBox*. Aceasta produce o casetă de dialog modală

dintr-o resursă de casetă de dialog șablon. *DialogBox* nu returnează controlul programului apelant până când funcția callback respectivă nu închide caseta de dialog modală în urma apelării funcției *EndDialog*. (Veți învăța mai mult despre *EndDialog* în secțiunea 1334.) Macrocomanda *DialogBox* utilizează funcția *DialogBoxParam* detaliată în secțiunea 1330. Prototipul macrocomenzii *DialogBox* este următorul:

```
int DialogBox(
HINSTANCE hInstance,     // identificator pentru instanta aplicatiei
LPCTSTR lpTemplate,      // identifica sablonul pentru caseta de
                         // dialog
HWND hWndParent,         // identificator pentru fereastra parinte
DLGPROC lpDialogFunc     // pointer la procedura casetei de dialog
);
```

Macrocomanda *DialogBox* acceptă patru parametri, prezentați în Tabelul 1326.

Parametru	Descriere
hInstance	Identifică o instanță a programului al cărui fișier executabil conține caseta de dialog șablon.
lpTemplate	Identifică șablonul casetei de dialog. Acest parametru este fie un pointer la un șir de caractere terminat în *NULL* care specifică numele casetei de dialog șablon, fie o valoare întreagă care precizează identificatorul resursei casetei de dialog șablon. Dacă parametrul specifică un identificator de resursă, cuvântul mai semnificativ trebuie să fie zero, iar cel mai puțin semnificativ trebuie să conțină identificatorul. Pentru a crea această valoare, puteți utiliza macrodefiniția *MAKEINTRESOURCE*.
hWndParent	Identifică fereastra care deține caseta de dialog.
lpDialogFunc	Indică procedura casetei de dialog. Secțiunea 1327 explică pe larg funcția callback *DialogProc*.

***Tabelul 1326** Parametrii macrocomenzii **DialogBox**.*

Dacă apelul la *DialogBox* reușește, macrocomanda returnează parametrul *nResult* în apelul funcției *EndDialog* utilizată pentru închiderea casetei de dialog. Dacă macrocomanda eșuează, ea returnează valoarea –1.

Macrocomanda *DialogBox* utilizează funcția *CreateWindowEx* pentru a crea caseta de dialog. *DialogBox* trimite atunci un mesaj *WM_INITDIALOG* către procedura casetei de dialog (împreună cu un mesaj *WM_SETFONT*, dacă șablonul menționează stilul *DS_SETFONT*). Funcția afișează caseta de dialog (indiferent dacă șablonul specifică sau nu stilul *WS_VISIBLE*), dezactivează fereastra proprietar și începe propria buclă de mesaje pentru primirea și distribuirea mesajelor casetei de dialog.

Când procedura de prelucrare a mesajelor casetei de dialog apelează funcția *EndDialog*, atunci *DialogBox* distruge caseta de dialog, închide bucla de mesaje, activează fereastra proprietar (dacă a fost anterior activată) și returnează parametrul *nResult* către fereastra apelantă.

CD-ROM-ul care însoțește cartea de față include programul *dlgbox.cpp* care produce o casetă de dialog în care utilizatorul poate introduce un număr întreg și un șir de caractere. Rețineți că programul declară întregul și șirul de caractere ca variabile globale, pe care programul le poate schimba din bucla de procesare a casetei de dialog. Următorul fragment de cod din *dlgbox.cpp* cuprinde funcția *WndProc* care produce caseta de dialog:

```
Case IDM_TEST :
   DialogBox(hInst, "TestDialog", hWnd,
      (DLGPROC )(TestDialogProc) );
   break;
```

După compilarea și executarea programului *dlgbox.cpp* ecranul va afișa următoarea ieșire:

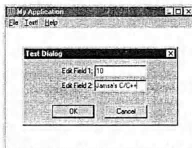


Figura 1326 *Ieșirea programului* ***dlgbox.cpp***.

BUCLA DE MESAJE A CASETEI DE DIALOG

C/C++1327

Așa cum ați învățat, când programul dumneavoastră creează casete de dialog, ele vor utiliza propria procedură de mesaje și nu procedura *WndProc* utilizată de ferestrele părinte. Programele dumneavoastră pot avea multe proceduri de mesaj diferite, chiar una pentru fiecare casetă de dialog. Puteți denumi procedura de mesaj cu orice nume doriți pentru că transmiteți adresa procedurii de fiecare dată când creați casete de dialog. De exemplu, în secțiunea 1326, programul *dlgbox.cpp* produce o casetă de dialog cu următorul apel la macrocomanda *DefineBox:*

```
DialogBox(hInst, "TestDialog", hWnd,
(DLGPROC)TestDlgProc );
```

Ultimul parametru, pointerul la procedura de mesaje, comunică sistemului Windows unde trebuie să trimită mesajele destinate casetei de dialog. Așa cum vedeți în următorul fragment de cod, procedura de mesaje pentru caseta de dialog operează prelucrări asemănătoare cu procedura *WndProc:*

```
LRESULT CALLBACK TestDlgProc( HWND hDlg, UINT uMsg,
            WPARAM wParam, LPARAM lParam )
{
   switch( uMsg )
   {
    case WM_INITDIALOG :
          SetDlgItemInt ( hDlg, IDC_EDIT1, nEditOne, TRUE );
          SetDlgItemText( hDlg, IDC_EDIT2, szEditTwo );
          break;
    case WM_COMMAND :
```

```
            switch( LOWORD( wParam ) )
            {
              case IDOK :
                {
                  BOOL bTran;
                  nEditOne = GetDlgItemInt( hDlg, IDC_EDIT1,
    &bTran, TRUE );
                  GetDlgItemText( hDlg, IDC_EDIT2, szEditTwo,
  sizeof(szEditTwo)-1 );
                  EndDialog( hDlg, IDOK );
                }
                break;

              case IDCANCEL :
                EndDialog( hDlg, IDCANCEL );
                break;
            }
            break;
        default :
            return( FALSE );
     }
     return( TRUE );
  }
```

În cazul procedurii de mesaje a casetei de dialog prezentată în fragmentul de cod de mai sus, procedura testează două mesaje de bază: *WM_INITDIALOG* și *WMCOMMAND*. Dacă procedura primește mesajul *WM_INITDIALOG*, ea va inițializa controalele pe care le va afișa caseta de dialog. Dacă în schimb, procedura primește mesajul *WM_COMMAND*, ea va testa cuvântul mai puțin semnificativ al valorii *wParam*. Așa cum știți, cuvântul mai puțin semnificativ al valorii *wParam* deține constanta comenzii selectate de utilizator. Dacă utilizatorul execută clic pe OK (și *wParam* conține constanta *IDOK*), procedura va salva valorile din casetele de editare în două variabile globale. Dacă utilizatorul execută click pe *Cancel* (și *wParam* conține constanta *IDCANCEL*), procedura va închide caseta de dialog fără să salveze vreo eventuală modificare făcută de utilizator.

1328 MAI MULTE DESPRE MANIPULAREA CONTROALELOR



În secțiunea 1327 ați învățat despre procedura de mesaje a casetei de dialog care tratează mesajele pe care Windows le repartizează casetei de dialog. În cazul programului *dlgbox.cpp*, procedura de mesaje inițializează controalele și, în funcție de intrările de la utilizator, salvează fiecare valoare a controlului în variabila globală. Procedura de mesaje utilizează funcțiile *SetDlgItemInt* și *SetDlgItemText* pentru a inițializa controalele ferestrei și funcțiile *GetDlgItemInt* și *GetDlgItemText* pentru a prelua valorile. Veți utiliza frecvent aceste funcții în programele dumneavoastră pentru a inițializa controalele și pentru a prelua valorile acestora. Prototipurile acestor funcții sunt prezentate mai jos:

```
BOOL SetDlgItemInt (HWND hDlg, int nIDDlgItem, UINT uValue, BOOL
                   bSigned);
BOOL SetDlgItemText(HWND hDlg, int nIDDlgItem, LPCTSTR lpString);
UINT GetDlgItemInt(HWND hDlg, int nIDDlgItem, BOOL *lpTranslated,
                   BOOL bSigned);
UINT GetDlgItemText(HWND hDlg, int nIDDlgItem, LPTSTR lpString,
                    int nMaxCount);
```

Rețineți că ambele funcții *Set* returnează o valoare *BOOL* care indică reușita sau eșecul lor. Funcțiile *Get* returnează o valoare *UINT* (unsigned integer). Valoarea returnată de funcția *GetDlgItemText* indică numărul de caractere pe care funcția le copiază în bufferul *lpString*. Tabelul 1328 listează parametrii celor patru funcții și arată cărei funcții îi revine fiecare parametru.

Parametru	Funcții	Descriere
hDlg	*SetDlgItemInt* *SetDlgItemText* *GetDlgItemInt* *GetDlgItemText*	Identifică acea casetă de dialog care conține controlul.
nIDDlgItem	*SetDlgItemInt* *SetDlgItemText*	Specifică ce control urmează a fi modificat.
	GetDlgItemInt *GetDlgItemText*	Specifică acel control din care se obține valoarea.
uValue	*SetDlgItemInt*	Specifică valoarea întreagă utilizată pentru generarea elementului de text.
bSigned	*SetDlgItemInt* *GetDlgItemInt*	Specifică dacă parametrul *uValue* este cu semn sau fără semn. Pentru *GetDlgItemInt*, specifică dacă valoarea returnată este cu semn sau fără semn. Dacă acest parametru este TRUE (adevărat), *uValue* este cu semn. Dacă parametrul este TRUE și *uValue* este mai mic ca zero, *SetDlgItemInt* plasează semnul minus înaintea primei cifre a șirului. Dacă parametrul este FALSE, *uValue* este fără semn.
lpString	*SetDlgItemText*	Pentru funcția *SetDlgItemText*, *lpString* specifică șirul *GetDlgItemText* ce va fi plasat în control. Pentru funcția *GetDlgItemText*, parametrul specifică șirul buffer în care funcția va plasa valoarea returnată a controlului.
lpTranslated	*GetDlgItemInt*	Indică valoarea de tip *Boolean* care primește valoarea de reușită sau eșec a funcției. *True* semnifică reușita. *False* semnifică eșecul. Acest parametru este opțional pentru că poate fi *NULL*. Când este *NULL*, funcția nu returnează nici o informație despre reușită sau eșec.
nMaxCount	*GetDlgItemText*	Specifică limita numărului de caractere pe care funcția îl va citi din control în șirul buffer *lpString*.

Tabelul 1328 *Parametrii funcțiilor* ***SetDlgItemInt***, ***SetDlgItemText***, ***GetDlgItemInt***, ***GetDlgItemText***.

1329 MACROCOMANDA CREATEDIALOG



În secțiunea 1326 ați utilizat macrocomanda *DialogBox* din resursa șablonului de casetă de dialog. Așa cum ați învățat, programul dumneavoastră va crea deseori casete de dialog nemodale, în plus față de cele modale. Puteți utiliza macrocomanda *CreateDialog* pentru a construi casete de dialog nemodale. Această macrocomandă produce o casetă de dialog nemodală dintr-o resursă a casetei de dialog șablon. Macrocomanda *CreateDialog* utilizează funcția *CreateDialogParam*, prezentată în detaliu în secțiunea 1330. Veți implementa macrocomanda *CreateDialog* în acord cu forma generală prezentată mai jos:

```
HWND CreateDialog(
    HINSTANCE hInstance,  // identificator pentru instanta
                          // aplicatiei
    LPCTSTR lpTemplate,   // identifica numele sablonului caseta
                          // de dialog
    HWND hWndParent,      // identificator pentru fereastra
                          // proprietar
    DLGPROC lpDialogFunc // pointer la procedura casetei de
                          // dialog
);
```

Așa cum vedeți, macrocomanda *CreateDialog* acceptă patru parametrii. Tabelul 1326 explică pe larg acești parametri.

Macrocomanda *CreateDialog* utilizează funcția *CreateWindowEx* pentru a crea o casetă de dialog. Imediat după apelarea ei, *CreateDialog* trimite un mesaj *WM_INITDIALOG* (împreună cu un mesaj *WM_SETFONT* dacă șablonul specifică stilul *DS_SETFONT*) către procedura casetei de dialog. Funcția va afișa caseta de dialog dacă șablonul specifică stilul *WS_VISIBLE*. În sfârșit, *CreateDialog* returnează identificatorul ferestrei casetei de dialog.

După ce *CreateDialog* revine din prelucrare, aplicația utilizează funcția *ShowWindow* pentru a afișa caseta de dialog (dacă nu este deja afișată). Aplicația utilizează funcția *DestroyWindow* pentru a elimina caseta de dialog. Pentru a înțelege mai bine procedura executată de macrocomanda *CreateDialog*, să analizăm programul *Create_Dialog.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Când utilizatorul selectează opțiunea *Test!*, funcția *WndProc* produce o casetă simplă nemodală, așa cum se prezintă în următorul fragment de cod:

```
Case IDM_TEST :
  if (!hDlgModeless)
    hDlgModeless = CreateDialog(hInst, "TestDialog", hWnd,
        (DLGPROC )(TestDialogProc) );
  break;
```

Când utilizatorul selectează opțiunea *Test!*, programul *Create_Dialog* testează dacă acea casetă de dialog nemodală este în acel moment deschisă. Dacă este, programul nu prelucrează nimic. Dacă nu este deschisă, *Create_Dialog* va utiliza macrocomanda *CreateDialog* pentru a afișa caseta de dialog nemodală.

FUNCȚIA CREATEDIALOGPARAM

C/C++1330

În secțiunea 1320 ați învățat cum să utilizați macrocomanda *DialogBox* pentru a crea casete de dialog modale. În secțiunea 1329 ați învățat cum este utilizată în programul dumneavoastră macrocomanda *CreateDialog* pentru a crea casete de dialog nemodale. Când programul dumneavoastră trebuie să creeze casete de dialog nemodale, este posibilă de asemenea apelarea funcției *CreateDialogParam*. Funcția *CreateDialogParam* (invocată de macrocomanda *CreateDialog*, ca parte din prelucrarea sa) creează o casetă de dialog nemodală dintr-o resursă de șablon al casetei de dialog. Funcția *CreateDialogParam* permite, de asemenea, programului dumneavoastră să transmită o valoare definită de aplicație la procedura casetei de dialog, ca parametru *lParam din mesajul WM_INITDIALOG*. Aplicațiile pot utiliza parametrul *lParam* pentru a inițializa elementele de control ale casetei de dialog. Veți implementa funcția *CreateDialogParam* în programele dumneavoastră în prototipul descris mai jos:

```
HWND CreateDialogParam(
    HINSTANCE hInstance,      // identificator pentru instanta
                              // aplicatiei
    LPCTSTR lpTemplateName,  // identifica sablonul casetei de
                              // dialog
    HWND hWndParent,          // identificator pentru fereastra
                              // proprietar
    DLGPROC lpDialogFunc,     // pointer la procedura casetei de
                              // dialog
    LPARAM dwInitParam        // valoare de initializare
);
```

Funcția *CreateDialogParam* acceptă cinci parametri, asemănători cu cei definiți în Tabelul 1326, cu excepția ultimului parametru. Parametrul *dwInitparam* specifică valoarea care va fi transmisă procedurii casetei de dialog în parametrul *lParam* din mesajul *WM_INITDIALOG*.

Funcția *CreateDialogParam* utilizează pentru crearea casetei de dialog, funcția *CreateWindowEx*. *CreateDialogParam* trimite apoi un mesaj *WM_INITDIALOG* (împreună cu un mesaj *WM_SETFONT* dacă șablonul menționează stilul *DS_SETFONT*) către procedura casetei de dialog. Dacă șablonul specifică stilul *WS_VISIBLE*, funcția va afișa caseta de dialog. În final, dacă reușește, *CreateDialogParam* va returna identificatorul ferestrei casetei de dialog.

După ce *CreateDialogParam* returnează controlul, aplicația dumneavoastră va utiliza funcția *ShowWindow* pentru a afișa caseta de dialog (dacă nu este deja afișată). Aplicația va utiliza *DestroyWindow* pentru a elimina caseta de dialog. Pentru a înțelege mai bine cum operează *CreateDialogParam*, să analizăm programul *CreatePDialog.cpp* conținut în CD-ROM-ul care însoțește această carte. Următorul fragment de cod din funcția *WndProc* utilizează funcția *CreateDialogParam*:

```
Case IDM_TEST :
   if (!hDlgModeless)
     {
      hDlgModeless = CreateDialogParam(hInst, "TestDialog", hWnd,
                        (DLGPROC ) (TestDlgProc), (LPARAM)lpMem );
     }
   break;
```

Programul *CreatePDialog.cpp* definește structura de tip *DLGDATA* pentru a reține starea curentă a fiecărui buton în caseta de dialog, cum se prezintă mai jos:

```
typedef struct {
    BOOL bChecked;
    BOOL bRadio1;
}
    DLGDATA;
```

Când programul se inițializează (adică atunci când procedura *WndProc* primește mesajul *WM_CREATE*), el alocă un identificator pentru instanța structurii. Când utilizatorul selectează *Test!* programul transmite identificatorul ca parametru ultim, în programul *CreatePDialog.cpp*. Procedura *TestDlgProc* primește identificatorul ca parametru *lParam* și utilizează valorile pentru a inițializa starea controalelor din dialog.

1331 PRELUCRAREA IMPLICITĂ A MESAJELOR ÎN CASETA DE DIALOG

C/C++

Așa cum ați învățat, programul dumneavoastră trebuie să creeze o funcție de prelucrare a mesajelor casetei de dialog pentru tratarea mesajelor pe care Windows le repartizează casetelor de dialog. Așa cum ați învățat, funcția de prelucrare a mesajelor casetei de dialog execută aceleași etape ca și funcția *WndProc* care tratează mesajele de fereastră din programul dumneavoastră. Așa cum ați învățat despre funcția *WndProc*, programul dumneavoastră trebuie întotdeauna să definească funcția Windows de prelucrare implicită a mesajelor (*DefWindowProc*) ca ultim *case* în instrucțiunea *switch*, cum prezentăm mai jos:

```
default :
    return()DefWindowProc(hWnd, uMsg, wParam, iParam));
  }
```

Dacă definiți o clasă fereastră separată pentru a crea o fereastră casetă de dialog, trebuie de asemenea să definiți o funcție implicită de prelucrare a mesajelor pentru casetele de dialog, la fel ca și în fereastra de bază. Pentru a defini o funcție implicită de prelucrare a mesajelor pentru casetele de dialog, programul dumneavoastră trebuie să utilizeze funcția *DefDlgProc*. Această funcție realizează prelucrarea implicită a mesajelor pentru procedurile fereastră care aparțin unei clase de casete de dialog definite de aplicație. Veți implementa funcția *DefDlgProc* în programele dumneavoastră, astfel:

```
LRESULT DefDlgProc(
    HWND hDlg,    // identificator pentru caseta de dialog
    UINT Msg,       // mesaj
    WPARAM wParam,   // parametrul primului mesaj
    LPARAM lParam // parametrul celui de-al doilea mesaj
);
```

Funcția *DefDlgProc* stabilește procedura fereastră implicită a clasei casetă de dialog predefinită. Această procedură pune la dispoziție prelucrări interne pentru caseta de dialog prin trimiterea mesajelor la procedura casetei de dialog și realizarea tratării implicite a oricărui mesaj returnat cu valoarea *False* de către procedura casetei de dialog. Aplicațiile care produc proceduri fereastră personalizate pentru casetele de dialog personalizate utilizează deseori funcția *DefDlgProc* pentru a realiza prelucrarea implicită a mesajelor, în locul funcției *DefWindowProc*.

Aplicațiile creează casete de dialog personalizate prin completarea structurii *WNDCALSS* cu informațiile corespunzătoare și înregistrarea clasei cu funcția *RegisterClass*. Unele aplicații utilizează funcția *GetClassInfo* pentru a completa structura și specifica numele casetei de dialog predefinite. În aceste cazuri, înaintea înregistrării clasei, aplicațiile modifică cel puțin membrul *lpszClassName*. În toate cazurile, trebuie să stabiliți membrul *cbWndExtra* din *WNDCLASS* pentru o clasă casetă de dialog personalizată, cel puțin ca *DLGWINDOWEXTRA*.

Procedura unei casete de dialog nu trebuie să apeleze funcția *DefDlgProc*; procedând astfel se ajunge la executare recursivă. Pentru a înțelege mai bine prelucrările efectuate de funcția *DefDlgproc*, să analizăm programul *DefDlgP.cpp* conținut pe CD-ROM-ul care însoțește cartea de față.

Când analizați definiția funcției *WndProc*, rețineți că *parametrul WM_CREATE* inițializează clasa casetă de dialog. Definițiile de resurse *BlueDlg* fac din caseta de dialog o clasă separată, ca mai jos:

```
TESTDIALOG DIALOG DISCARDABLE 0, 0, 180, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
  WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
CLASS "BlueDlg"
BEGIN
    CHECKBOX "Check box control.",IDC_CHECKBOX,9,7,70,10
    GROUPBOX "Radio Buttons",-1,7,21,86,39
    RADIOBUTTON "First",IDC_RADIO1,13,32,37,10,WS_GROUP |
      WS_TABSTOP
    RADIOBUTTON "Second",IDC_RADIO2,13,45,39,10
    PUSHBUTTON "Done",IDCANCEL,116,8,50,14,WS_GROUP
END
```

Când executați programul *DefDlgP* și selectați opțiunea *Test!*, programul va utiliza clasa *BlueDlg* pentru a crea caseta de dialog. Pentru fiecare mesaj Windows pe care caseta de dialog nu îl tratează, *DefDlgP* va apela funcția implicită de prelucrarea mesajelor în caseta de dialog (în cazul programului *DefDlgP*, funcția implicită de prelucrarea mesajului este *TestDlgProc*).

UTILIZAREA FUNCȚIEI DLGDIRLIST PENTRU A CREA O CASETĂ DE DIALOG TIP LISTĂ

C/C++1332

În secțiunile precedente ați utilizat macrodefiniția *DialogBox*, funcția *CreateDialogParam* și funcția *CreateDialog* pentru a crea casete de dialog simple, care conțin unul sau mai multe controale.Veți folosi de multe ori casete de dialog în programele dumneavoastră pentru a afișa informații despre fișierele conținute de o dischetă sau o unitate de disc. Pentru a simplifica procesul de realizare a casetelor de dialog care conțin informații despre fișiere conținute pe disc, interfața Win32 API pune la dispoziție funcția *DlgDirList*. Programul va utiliza funcția *DlgDirList* pentru a completa o anumită casetă cu listă cu numele tuturor fișierelor care corespund căii sau numele de fișier specificate. Programul dumneavoastră va utiliza funcția *DlgDirList* cu următorul prototip:

```
int DlgDirList(
    HWND hDlg,           // identificator pentru caseta de dialog
                         // cu caseta lista
    LPTSTR lpPathSpec,   // pointer la sirul cu numele sau calea
                         // fisierului
    int nIDListBox,      // identificatorul casetei lista
    int nIDStaticPath,   // identificatorul controlului static
    UINT uFileType       // atribute de fisier pentru afisare
);
```

După cum vedeți, funcția *DlgDirList* acceptă cinci parametrii. Tabelul 1332.1 explică acești parametrii în detaliu.

Parametru	Descriere
hDlg	Identifică acea casetă de dialog care conține caseta listă.
lpPathSpec	Indică un șir terminat cu NULL care conține numele fișierului sau calea. *DlgDirList* modifică acest șir care trebuie să fie suficient de lung pentru a cuprinde modificările.
nIDListBox	Specifică identificatorul unei casete listă. Dacă parametrul este zero, *DlgDirList* consideră că nu există nici o casetă listă și nu încearcă să încarce vreuna.
nIDStaticPath	Specifică identificatorul controlului static utilizat pentru afișarea unității curente sau directorului. Dacă acest parametru este zero, DlgDirList consideră că nu este prezent un astfel de control.
uFileType	Specifică atributele fișierelor ce vor fi afișate. Acest parametru ia una sau mai multe din valorile prezentate în Tabelul 1332.2.

Tabelul 1332.1 *Parametrii funcției* ***DlgDirList***.

Așa cum prezintă Tabelul 1332.1 parametrul *uFileType* acceptă o serie de valori diferite. Când utilizați valorile pentru parametrul *uFileType*, veți folosi operatorul SAU pe bit pentru a combina valorile. Tabelul 1332.2 listează valorile acceptate ale parametrului *uFileType*:

Valoare	Descriere
DDL_ARCHIVE	Include fișiere arhivate.
DDL_DIRECTORY	Include subdirectoare. Subdirectoarele sunt incluse în paranteze drepte ([ ]).
DDL_DRIVES	Include unitățile de disc. Unitățile de disc sunt listate în forma [-s-], unde s este litera unității de disc.
DDL_EXCLUSIVE	Include numai fișiere cu atributele specificate. Implicit, sunt listate fișierele cu drept de citire-scriere chiar dacă nu este specificat *DDL_READWRITE*.
DDL_HIDDEN	Include fișiere ascunse.
DDL_READONLY	Include fișiere numai cu drept de citire.
DDL_READWRITE	Include fișiere cu drept de citire-scriere fără atribute suplimentare.
DDL_SYSTEM	Include fișiere sistem.

Valoare	Descriere
DDL_POSTMSGS	Expediază mesajele în coada de mesaje a aplicației. Implicit, *DlgDirList* trimite mesajele direct la procedura mesajelor casetei de dialog.

*Tabelul 1332.2 Valorile acceptate de parametrul **uFileType**.*

Dacă funcția *DlgDirList* afișează o listă – chiar dacă este o listă goală – funcția va returna o valoare diferită de zero. Dacă șirul introdus nu conține o valoare validă de cale (sau dacă o eroare oprește generarea listei), valoarea returnată va fi zero.

Dacă pentru parametrul *lpPathSpec* specificați un șir cu zero caractere sau un nume de director fără fișiere, funcția *DlgDirList* va schimba șirul cu „*.*". Parametrul *lpPathSpec* are următoarea formă:

```
[unitate_disc:] [[\u]director[\idirector]\u] [nume_fisier]
```

În acest exemplu, *unitate_disc* este o literă de unitate de disc, *director* este un nume valid de director și *nume_fisier*, un nume valid de fișier care conține cel puțin un caracter de înlocuire (? sau *). Dacă *lpPathSpec* include un nume de director sau de unitate de disc sau ambele, funcția va schimba unitatea și directorul curente cu unitatea și directorul specificate, înainte ca funcția să completeze caseta listă. Funcția, de asemenea, va actualiza controlul static identificat de parametrul *nIDStaticPath* cu numele unității sau directorului curente sau al ambelor.

După ce funcția completează lista casetă, *DlgDirList* actualizează *lpPathSpec* prin eliminarea unității sau directorului, sau a ambelor, din calea și numele de fișier. În acest moment *DlgDirList* trimite mesajele *LB_RESETCONTEXT* și *LB_DIR* către caseta listă. Pentru a înțelege mai bine cum operează funcția *DlgDirList*, să analizăm programul *List_Dir.cpp* din CD-ROM-ul care însoțește cartea de față. Fișierul *List_Dir.rc* definește caseta de dialog care cuprinde caseta de text static, caseta listă și butonul de comandă pentru închiderea dialogului, cum se prezintă mai jos:

```
TESTDIALOG DIALOG DISCARDABLE 20, 20, 150, 110
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
  WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT IDC_DIRECTORY,6,5,136,13,ES_AUTOHSCROLL |
      ES_READONLY |
NOT WS_TABSTOP
    LISTBOX IDC_LIST,6,20,136,59,LBS_SORT |
      LBS_NOINTEGRALHEIGHT | LBS_DISABLENOSCROLL | WS_VSCROLL |
      WS_TABSTOP
    PUSHBUTTON "Done",IDCANCEL,50,87,50,14,WS_GROUP
END
```

Fișierul *List_Dir.cpp* utilizează funcția *DlgDirList* pentru realizarea unei casete de dialog din *WndProc*, atunci când utilizatorul selectează opțiunea *Test!* Caseta de dialog conține o casetă listă care afișează conținutul directorului curent și o casetă de text static care afișează numele directorului. Dacă utilizatorul execută dublu-clic pe oricare fișier din caseta listă, programul

se comută la directorul sau unitatea respective, dacă selecția este un director sau o unitate de disc. Programul va afișa atunci selecția în caseta de text static, indiferent de tipul selecției. Când compilați și executați programul *List_Dir*, monitorul va afișa ieșirea prezentată în Figura 1332.

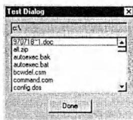


***Figura 1332** Ieșirea programului **List_Dir**.*

1333 RĂSPUNSURILE LA SELECȚIILE UTILIZATORULUI ÎN CASETĂ LISTĂ

C/C++

În secțiunea 1332 ați realizat programul *List_Dir* care afișează fișierele din directorul curent când programul afișează caseta de dialog. Funcția *TestDlgProc* din program modifică informațiile din caseta de text static când utilizatorul execută dublu-clic pe un articol din caseta listă.

În ultimele secțiuni ați învățat în detaliu despre captarea textului introdus de utilizator. În programul *List_Dir.cpp* însă, prelucrarea este relativ simplă: programul mai întâi testează dacă utilizatorul a executat dublu-clic. Dacă a procedat astfel, programul testează funcția *DlgDirSelectEx* pentru a determina selecția utilizatorului – un director sau o unitate de disc. Funcția *DlgDirSelectEx* preia selecția curentă dintr-o casetă listă cu o singură selecție. Funcția presupune că *DlgDirList* a completat caseta listă și că selecția este o literă de unitate de disc, un nume de fișier sau de director. Programele dumneavoastră vor implementa funcția *DlgDirSelectEx* cu prototipul prezentat mai jos:

```
BOOL DlgDirSelectEx(
   HWND hDlg,      // identificator pentru caseta de dialog cu
                   // caseta lista
   LPTSTR lpString, // pointer la bufferul cu sirul de cale
   int nCount,      // numarul de caractere din sirul de cale
   int nIDListBox    // identificatorul casetei lista
);
```

Dacă selecția curentă este un nume de director, funcția returnează o valoare diferită de zero. Dacă selecția curentă nu este un nume de director, funcția va returna valoarea zero. Funcția *DlgDirSelectEx* copiază selecția în bufferul la care indică parametrul *lpString*. Dacă selecția curentă este un nume de director sau o literă de unitate de disc, funcția *DlgDirSelectEx* va înlătura parantezele drepte (și liniuțele pentru literele de unități de disc) în așa fel încât numele sau litera respectivă să poată fi inserată într-o nouă cale de către funcția *DlgDirList*. Dacă nu există selecții, parametrul *lpString* nu se modifică.

Funcția *DlgDirSelectEx* trimite mesajele *LB_GETCURSEL* și *LB_GETTEXT* către caseta listă. Funcția nu permite casetei listă să returneze mai mult de un nume de fișier. Caseta listă nu trebuie să fie o casetă cu selecții multiple. Dacă ar fi o casetă listă cu selecții multiple, *DlgDirSelectEx* nu ar returna valoarea zero, iar *lpString* ar rămâne nemodificat. Când invocați funcția *DlgDirSelectEx*, programul dumneavoastră trebuie să testeze valoarea returnată și să răspundă corespunzător, ca funcția *TestDlgProc* de mai jos, din programul *List_Dir.cpp:*

```
LRESULT CALLBACK TestDlgProc( HWND hDlg, UINT uMsg,
                              WPARAM wParam, LPARAM lParam )
{
  static char szTmp[255];

  switch( uMsg )
   {
     case WM_INITDIALOG :
      DlgDirList( hDlg, "*.*", IDC_LIST, IDC_DIRECTORY,
           DDL_DIRECTORY | DDL_DRIVES );
      break;
     case WM_COMMAND :
      switch( LOWORD( wParam ) )
      {
       case IDC_LIST :
        if ( HIWORD( wParam ) == LBN_DBLCLK )
        {
         if (DlgDirSelectEx(hDlg, szTmp, sizeof(szTmp),
           IDC_LIST ) )
          {
           strcat( szTmp, "*.*" );
           DlgDirList(hDlg, szTmp,IDC_LIST,
             IDC_DIRECTORY, DDL_DIRECTORY | DDL_DRIVES );
          }
         else
           MessageBox( hDlg, szTmp, "File Selected",
             MB_OK | MB_ICONINFORMATION );
        }
        break;

       case IDCANCEL:
        EndDialog( hDlg, IDCANCEL );
        break;
      }
      break;
     default :
      return( FALSE );
   }

  return( TRUE );
}
```

1334 ÎNCHIDEREA CASETEI DE DIALOG

C/C++

Pe parcursul secțiunilor anterioare, ați învățat cum să realizați și să afișați diverse casete de dialog. Așa cum ați văzut în fragmentele de cod anterioare, trebuie întotdeauna să utilizați funcția *EndDialog* pentru a închide caseta de dialog modală. Funcția *EndDialog* distruge caseta de dialog modală, ceea ce impune sistemului să încheie toate procesele casetei de dialog. Veți utiliza funcția *EndDialog* în programele dumneavoastră, după prototipul descris mai jos:

```
BOOL EndDialog(
    HWND hDlg, // identificator pentru caseta de dialog
    int nResult // valoare pentru returnare
);
```

Parametrul *hDlg* identifică acea casetă de dialog pe care funcția *EndDialog* o va distruge. Parametrul *nResult* permite programului dumneavoastră să specifice valoarea care urmează a fi returnată aplicației din funcția care a produs caseta de dialog. Funcția *EndDialog* trebuie utilizată pentru a distruge casetele de dialog produse cu funcțiile *DialogBox, DialogBoxParam, DialogBoxIndirect* și *DialogBoxIndirectParam*. Aplicația apelează funcția *EndDialog* din procedura casetei de dialog. Această funcție nu trebuie utilizată în alte scopuri.

Procedura casetei de dialog poate apela oricând funcția *EndDialog*, chiar pe parcursul tratării mesajului *WM_INITDIALOG*. Dacă aplicația dumneavoastră apelează *EndDialog* în timp de aplicația tratează mesajul *WM_INITDIALOG*, Windows va distruge caseta de dialog înainte de a o afișa și ca Windows să stabilească focusul intrărilor în caseta respectivă.

EndDialog nu distruge caseta de dialog imediat. Ea stabilește un indicator și permite procedurii casetei de dialog să returneze controlul sistemului. Sistemul testează indicatorul înainte de a încerca să preia următorul mesaj din coada aplicației. Dacă *EndDialog* a stabilit anterior indicatorul, sistemul va închide bucla de mesaje, va distruge caseta de dialog și va folosi valoarea parametrului *nResult* ca valoare returnată din funcția care a creat caseta de dialog.

1335 INTRĂRILE DE LA UTILIZATOR

C/C++

În secțiunile anterioare ați creat diverse tipuri de programe Windows. Fiecare program, însă, are o singură trăsătură în comun cu oricare alt program: fiecare program primește intrări de la utilizator pentru a executa diverse operații, chiar dacă intrarea de la utilizator a fost o simplă comandă de închidere. Așa cum ați învățat, marea majoritate a programelor Windows pe care le veți scrie (cu excepția programelor de server automat) vor cere anumite tipuri de intrări de la utilizator pentru a executa operații utile. Programele de server automat sunt o excepție de reținut, întrucât intrările lor provin de la alte programe și nu de la utilizator.

În Windows, programele primesc intrări de la utilizator prin oricare din diversele sale dispozitive. Totuși, cele mai utilizate dispozitive sunt mouse-ul și tastatura. Alte dispozitive de intrare sunt creionul (lightpen), ecranul senzitiv, joystick și altele. De fiecare dată când utilizatorul apasă o tastă (și de fiecare dată când execută clic cu mouse-ul sau îl deplasează), Windows generează un mesaj. Acest mesaj poate ajunge la sistemul de operare, poate ajunge la fereastra programului dumneavoastră sau la caseta de dialog. În anumite cazuri, mesajul poate ajunge la locații multiple.

În Windows, programele dumneavoastră vor utiliza în general controale predefinite pentru a răspunde într-o manieră adecvată la evenimentele de la tastatură sau mouse, controale ca butoanele, controalele de editare și meniurile. În multe din următoarele secțiuni veți învăța despre metode suplimentare pe care le pot utiliza programele pentru a controla mai bine răspunsul la intrările de la tastatură sau mouse.

RĂSPUNSUL LA EVENIMENTELE GENERATE DE MOUSE

C/C++1336

Așa cum ați învățat, programele dumneavoastră vor răspunde la intrările de la utilizator în forma mesajelor de sistem. În secțiunile anterioare, programele dumneavoastră au testat mesajul *WM_COMMAND* pentru a determina dacă utilizatorul a trimis o comandă către programul dumneavoastră, dacă se răspunde la meniu sau selecția de control care a generat acea comandă. Când programul lucrează cu operații induse de mouse, programul testează mesajele Windows descrise în Tabelul 1336.

Mesaj	Semnificație
WM_CAPTURECHANGED	Windows trimite mesajul *WM_CAPTURECHANGED* către fereastra care pierde captarea mouse-ului.
WM_LBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din stânga al mouse-ului.
WM_LBUTTONDOWN	Utilizatorul a apăsat butonul din stânga al mouse-ului și îl ține apăsat.
WM_LBUTTONUP	Utilizatorul a eliberat butonul din stânga al mouse-ului.
WM_MBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din mijloc al mouse-ului (numai la mouse-urile cu trei butoane).
WM_MBUTTONDOWN	Utilizatorul a apăsat butonul din mijloc al mouse-ului (numai la mouse-urile cu trei butoane).
WM_MBUTTONUP	Utilizatorul a eliberat butonul din mijloc al mouse-ului (numai la mouse-urile cu trei butoane).
WM_MOUSEACTIVATE	Utilizatorul a executat clic pe mouse în fereastra inactivă la acel moment.
WM_MOUSEMOVE	Utilizatorul a deplasat mouse-ul în fereastră.
WM_NCLBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din stânga al mouse-ului în zona non-client a ferestrei.
WM_NCLBUTTONDOWN	Utilizatorul a apăsat butonul din stânga al mouse-ului în zona non-client a ferestrei și îl ține apăsat.
WM_NCLBUTTONUP	Utilizatorul a eliberat butonul din stânga al mouse-ului în zona non-client a ferestrei.
WM_NCMBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din mijloc al mouse-ului în zona non-client a ferestrei (numai la mouse-urile cu trei butoane).
WM_NCMBUTTONDOWN	Utilizatorul a apăsat butonul din mijloc al mouse-ului în zona non-client a ferestrei și îl ține apăsat (numai la mouse-urile cu trei butoane).

(continuare)

Mesaj	Semnificație
WM_NCMBUTTONUP	Utilizatorul a eliberat butonul din mijloc al mouse-ului în zona non-client a ferestrei (numai la mouse-urile cu trei butoane).
WM_NCMOUSEMOVE	Utilizatorul a deplasat mouse-ul în zona non-client a ferestrei.
WM_NCRBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din dreapta al mouse-ului în zona non-client a ferestrei.
WM_NCRBUTTONDOWN	Utilizatorul a apăsat butonul din dreapta al mouse-ului în zona non-client a ferestrei și îl ține apăsat.
WM_NCRBUTTONUP	Utilizatorul a eliberat butonul din dreapta al mouse-ului în zona non-client a ferestrei.
WM_RBUTTONDBLCLK	Utilizatorul a apăsat de două ori butonul din dreapta al mouse-ului.
WM_RBUTTONDOWN	Utilizatorul a apăsat butonul din dreapta al moûse-ului și îl ține apăsat.
WM_RBUTTONUP	Utilizatorul a eliberat butonul din dreapta al mouse-ului.

***Tabelul 1336** Mesajele de mouse ale sistemului Windows.*

Când o aplicație primește un mesaj de mouse, valoarea *lParam* conține pozițiile X și Y pe ecran ale cursorului. Windows stochează poziția Y în cuvântul mai semnificativ al lui *lParam*, iar poziția X în cuvântul semnificativ. Programul dumneavoastră trebuie să folosească instrucțiunile macro *LOWORD* și *HIGHWORD* pentru a extrage cele două valori. Secțiunea 1337 examinează mesajul *WM_MOUSEMOVE* pentru exemplificarea unui mesaj de mouse în Windows și prezintă un fragment de cod în care se vede cum se extrag valorile din parametrul *lParam*.

1337 UTILIZAREA MESAJULUI WM_MOUSEMOVE C/C++

Așa cum ați învățat, Windows generează mesaje diferite către aplicațiile dumneavoastră când utilizatorul manipulează mouse-ul. Unul din cele mai obișnuite mesaje pe care Windows le trimite programului este *WM_MOUSEMOVE*. Windows transmite mesajul *WM_MOUSEMOVE* către fereastră de fiecare dată când cursorul se mișcă. Dacă fereastra nu a captat anterior mouse-ul, Windows transmite mesajul *WM_MOUSEMOVE* către fereastra care conține cursorul. Altfel, Windows transmite mesajul *WM_MOUSEMOVE* către fereastra care a captat mouse-ul. Când programul primește mesajul *WM_MOUSEMOVE*, el trebuie să testeze următoarele valori înainte de a începe prelucrarea răspunsului:

```
fwKeys = wParam;            //indicatoare de taste
xPos = LOWORD(lParam);      //pozitia orizontala cursor
yPos = HIGHWORD(lParam);    //pozitia verticala cursor
```

Așa cum ați învățat în secțiunea 1336, parametrul *lParam* conține pozițiile X și Y ale cursorului. Parametrul *wParam* va conține valorile *fwKeys*. Valoarea *fwKeys* indică dacă au fost acționate taste virtuale. Valoarea *fwKeys* poate fi orice combinație prezentată în Tabelul 1337.

Valoare	Descriere
MK_CONTROL	Fixat dacă tasta CTRL este apăsată
MK_LBUTTON	Fixat dacă butonul din stânga al mouse-ului e apăsat
MK_MBUTTON	Fixat dacă butonul din mijloc al mouse-ului e apăsat
MK_RBUTTON	Fixat dacă butonul din dreapta al mouse-ului e apăsat
MK_SHIFT	Fixat dacă tasta SHIFT e apăsată

Tabelul 1337 *Valorile acceptate de parametrul* ***fwKeys***.

Pentru a capta coordonatele mouse-ului în procedura de prelucrare a mesajului, programul dumneavoastră trebuie să utilizeze un cod asemănător cu fragmentul de mai jos:

```
Case WM_MOUSEMOVE :
   {
     nXPos = LOWORD(lParam);
     nYPos = HIGHWORD(lParam);
     // alte instructiuni
```

În fragmentul precedent programul alocă poziția curentă a mouse-ului la valorile *nXPos* și *nYPos* de fiecare dată când utilizatorul deplasează mouse-ul.

CITIREA BUTOANELOR MOUSE-ULUI

C/C++1338

În secțiunea 1336, ați învățat cum să utilizați mesajul fereastră *WM_MOUSEMOVE* pentru a da posibilitatea programelor dumneavoastră să răspundă la mișcarea mouse-ului pe ecran. De asemenea, operația de adăugare a codului la program pentru a-l face să răspundă la acționarea butoanelor mouse-ului este la fel de simplă. De exemplu, dacă vreți ca programul să răspundă când utilizatorul execută dublu clic pe mouse oriunde în zona client a ferestrei, programul dumneavoastră va utiliza în funcția *WndProc* codul similar cu cel de mai jos:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                         LPARAM lParam )
  {
   switch( uMsg )
     {
       case WM_LBUTTONDBLCLK :
         MessageBox (hWnd, "Dublu clic pe mouse", NULL, MB_OK);
         break;
```

Dacă, pe de altă parte, doriți ca programul să răspundă diferit în cazul în care utilizatorul ține tasta CTRL apăsată în timp ce apasă butonul mouse-ului, puteți construi codul de mai jos:

```
   case WM_LBUTTONDBLCLK :
   if ((wParam && MK_CONTROL) == MK_CONTROL)
     {
      MessageBox (hWnd, "Dublu clic pe mouse cu tasta Control",
NULL, MB_OK);
      break;
     }
```

```
    else
      {
        MessageBox (hWnd, "Dublu clic pe mouse fara tasta
Control", NULL, MB_OK);
        break;
      }
```

Așa cum vedeți din codul de mai sus, programul poate testa dacă utilizatorul apasă pe o tastă virtuală prin aplicarea operației ȘI pe bit la valoarea *wParam* și la valoarea tastei virtuale pe care o testați. Pe măsură ce programul devine mai complex, veți proceda frecvent la testarea tastelor virtuale când prelucrați evenimentele de la mouse. Windows transmite numai anumite taste virtuale în parametrul *wParam* împreună cu un clic de mouse. Windows acceptă un mare număr de taste virtuale de care veți afla în secțiunea 1340.

1339 RĂSPUNSUL LA EVENIMENTELE DE LA TASTATURĂ



Așa cum ați învățat, Windows va expedia mesajele către aplicație de fiecare dată când utilizatorul execută o acțiune cu mouse-ul. Windows va repartiza, de asemenea, mesajele către aplicație de fiecare dată când utilizatorul acționează o tastă pe tastatura calculatorului. Mesajele expediate de Windows sunt listate în Tabelul 1339.

Mesaj	Descriere
WM_ACTIVATE	Evenimentul de la tastatură are loc în fereastra inactivă
WM_CHAR	Codul ASCII al literei, dacă utilizatorul a apăsat un caracter pe tastatură
WM_GETHOTKEY	Permite programului să preia „tasta caldă" anterior stabilită pentru fereastră
WM_HOTKEY	Utilizatorul a apăsat tasta *caldă*
WM_KEYDOWN	Utilizatorul a acționat o tastă
WM_KEYUP	Utilizatorul a eliberat o tastă apăsată
WM_KILLFOCUS	Trimis ferestrei imediat înainte ca fereastra să piardă focusul intrărilor de la tastatură
WM_SETFOCUS	Trimis ferestrei imediat după ce fereastra primește focusul intrărilor de la tastatură
WM_SETHOTKEY	Permite programului să stabilească o „tastă caldă" a ferestrei
WM_SYSCHAR	Codul ASCII al literei pe care utilizatorul a apăsat-o concomitent cu tasta ALT
WM_SYSKEYDOWN	Utilizatorul a apăsat o tastă concomitent cu tasta ALT
WM_SYSKEYUP	Utilizatorul a eliberat o tastă concomitent cu eliberarea tastei ALT

Tabelul 1339 *Mesajele de sistem pe care Windows le trimite pentru prelucrarea evenimentelor de la tastatură.*

Funcția *TranslateMessage* din bucla de mesaje a firului de execuție va genera mesajul *WM_CHAR* dacă recunoaște caracterul ca fiind un caracter ASCII. În general, aplicațiile vor

utiliza mesajul *WM_KEYDOWN* pentru a verifica tastele funcționale, tastele cursor, tastatura numerică și tastele de editare cum sunt PAGEUP și PAGEDOWN. Aceste taste folosesc din plin codurile de taste virtuale despre care veți învăța în următoarea secțiune.

Pe de altă parte, programul va utiliza *WM_CHAR* pentru a prelua litere, numere sau simboluri tipăribile. Utilizarea mesajului *WM_CHAR* pentru a prelucra evenimentele de la tastatură este cea mai indicată pentru că ASCII atribuie valori diferite literelor mici și celor mari. Cu *WM_KEYDOWN* aplicația dumneavoastră trebuie să verifice ce caracter a introdus utilizatorul, cât și actuala stare a tastei SHIFT. Mesajul *WM_CHAR* tratează tasta SHIFT.

Dacă utilizatorul apasă tasta ALT concomitent cu o altă tastă, aplicația va primi următoarea secvență de mesaj: *WM_SYSKEYDOWN, WM_SYSCHAR, WM_SYSKEYUP.* Programul dumneavoastră trebuie să utilizeze aceste mesaje pentru a testa secvențele specifice ale tastei ALT.

TASTELE VIRTUALE

C/C++ 1340

Așa cum ați învățat în secțiunea 1339, programele dumneavoastră pot prelucra un eveniment de la tastatură ca pe un caracter ASCII, dacă utilizează mesajul *WM_CHAR.* De multe ori însă, programele dumneavoastră sau utilizatorii vor avea nevoie de alte taste decât tastele ASCII normale cum ar fi, de exemplu, tastele cu săgeți, tastele numerice și altele. Când operați cu astfel de taste, trebuie să utilizați valorile tastelor virtuale. Tastele virtuale vă vor elibera de sarcina de a afla ce tip de tastatură deține utilizatorul, deoarece codul tastei virtuale pentru prima tastă funcțională trebuie să fie întotdeauna același, indiferent de modelul de fabricație al tastaturii. Mesajele *WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN* și *WM_SYSKEYUP* trimit codurile tastelor virtuale ca valoare *wParam* a mesajului.

În cadrul structurii codului de tastă virtuală, tastele numerice din minitastatura numerică dețin propriile lor coduri de taste virtuale. În plus, codul virtual al tastelor cu caractere sau numerice este echivalentul lor în majuscule ASCII (cu alte cuvinte *VK_A* este atât a cât și A). În sfârșit, rețineți că ambele taste SHIFT generează același cod de tastă virtuală. Tabelul 1340 listează codurile de taste virtuale definite în Win32 API.

Cod tastă	Valoare (hexazecimală)	Echivalent mouse sau tastatură
VK_LBUTTON	01	Butonul din stânga al mouse-ului
VK_RBUTTON	02	Butonul din dreapta al mouse-ului
VK_CANCEL	03	Utilizat pentru executare unui control-break
VK_MBUTTON	04	Buton de mouse din mijloc (mouse cu 3 butoane) sau butoanele din stânga și dreapta ale mouse-ului apăsate simultan
VK_BACK	08	Tasta Backspace
VK_TAB	09	Tasta Tab
VK_CLEAR	0C	Tasta Clear
VK_RETURN	0D	Tasta Enter
VK_SHIFT	10	Tasta Shift
VK_CONTROL	11	Tasta Ctrl
VK_MENU	12	Tasta Alt

(continuare)

Cod tastă	Valoare (hexazecimală)	Echivalent mouse sau tastatură
VK_PAUSE	13	Tasta Pause
VK_CAPITAL	14	Tasta Cap Lock
VK_ESCAPE	18	Tasta Escape
VK_SPACE	20	Bara de spațiu
VK_PRIOR	21	Tasta Page Up
VK_NEXT	22	Tasta Page Down
VK_END	23	Tasta End
VK_HOME	24	Tasta Home
VK_LEFT	25	Tasta cu săgeată la stânga
VK_UP	26	Tasta cu săgeată în sus
VK_RIGHT	27	Tasta cu săgeată la dreapta
VK_DOWN	28	Tasta cu săgeată în jos
VK_SELECT	29	Tasta Select
VK_EXECUTE	2B	Tasta Execute
VK_SNAPSHOT	2C	Tasta Printscreen
VK_INSERT	2D	Tasta Insert
VK_DELETE	2E	Tasta Delete
VK_HELP	2F	Tasta Help
VK_0_VK_9	30-39	Tastele 0-9
VK_A_VK_Z	41-5A	Tastele A-Z
VK_NUMPAD0	60	Tasta numerică 0
VK_NUMPAD1	61	Tasta numerică 1
VK_NUMPAD2	62	Tasta numerică 2
VK_NUMPAD3	63	Tasta numerică 3
VK_NUMPAD4	64	Tasta numerică 4
VK_NUMPAD5	65	Tasta numerică 5
VK_NUMPAD6	66	Tasta numerică 6
VK_NUMPAD7	67	Tasta numerică 7
VK_NUMPAD8	68	Tasta numerică 8
VK_NUMPAD9	69	Tasta numerică 9
VK_MULTIPLY	6A	Tasta Înmulțire
VK_ADD	6B	Tasta Adunare
VK_SEPARATOR	6C	Tasta Separator
VK_SUBTRACT	6D	Tasta Scădere

Cod tastă	Valoare (hexazecimală)	Echivalent mouse sau tastatură
VK_DECIMAL	6E	Tasta Zecimal
VK_DIVIDE	6F	Tasta Împărţire
VK_F1_VK_F24	70-87	Taste funcţionale F1-F24
VK_NUMLOCK	90	Tasta Num Lock
VK_SCROLL	91	Tasta Scroll Lock

Tabelul 1340 *Codurile tastelor virtuale definite în Win32 API.*

UTILIZAREA TASTELOR VIRTUALE

C/C++ 1341

Aşa cum aţi învăţat, programele dumneavoastră vor primi unul din cele trei tipuri de intrări de la tastatură, recunoscute de Windows ca intrări de caractere ASCII (în valoarea *wParam* a mesajului *WM_CHAR*); nerecunoscute, intrare caracter non-sistem în parametrul *wParam* a mesajului *WM_KEYDOWN* şi intrare caracter de sistem în parametrul *wParam* al mesajului *WM_SYSCHAR* sau al mesajului *WM_SYSKEYDOWN*. Aţi învăţat, de asemenea, cum să prelucraţi uşor intrările de la mouse în programele dumneavoastră. Prelucrarea mesajelor de la tastatură este la fel de uşoară. De exemplu, următorul fragment de cod va afişa caractere recunoscute într-o casetă de editare – când caracterul nerecunoscut este o tastă funcţională, codul va afişa o casetă de mesaj cu următorul conţinut:

```
LRESULT CALLBACK DlgTestProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                              LPARAM lParam )
  {
   switch( uMsg )
     {
      case WM_CHAR :
      GetDlgItem(hWnd, IDC_EDITBOX);
      lstrcat(szEditBox, wParam);
      SetDlgItemText(hWnd, IDC_EDITBOX, szEditBox);
      break;

     case WM_KEYDOWN :
      switch(wParam)
        {
         case VK_F1 :
          MessageBox(hWnd, "Actionat tasta functionala F1",
            NULL, MB_OK);
          break;
         case VK_F2 :
          MessageBox(hWnd, "Actionat tasta functionala F2",
  NULL, MB_OK);
          break;
// alte taste functionale virtuale
       }
```

Aşa cum vedeţi, programul verifică valorile tastelor funcţionale virtuale pe care Windows le stochează în parametrul *wParam* pentru a determina ce tastă virtuală a acţionat utilizatorul.

1342 UTILIZAREA MESAJULUI WM_KEYDOWN



Aşa cum aţi învăţat, Windows transmite mesajul *WM_KEYDOWN* către fereastra care posedă focusul intrărilor de la tastatură, când utilizatorul a acţionat o tastă non-sistem. O tastă non-sistem este o tastă acţionată de utilizator fără ca, simultan, să acţioneze şi tasta ALT. Aşa cum aţi învăţat în secţiunea 1341, Windows transmite codul tastei funcţionale acţionate de utilizator în parametrul *wParam*. Dar Windows transmite şi valoarea *lKeyData* în parametrul *lParam*. Parametrul *lKeyData* specifică numărul de repetări, codul de scanare, indicatorul de tastă extinsă, codul de context, indicatorul cu starea anterioară a tastei şi indicatorul cu starea de tranziţie, cum se prezintă în Tabelul 1342.

Valoare	Descriere
0-15	Specifică numărul de repetări. Valoarea este numărul de acţionări repetate ale tastei, ca rezultat al apăsării continue a tastei.
16-23	Specifică un cod de scanare. Valoarea depinde de fabricantul de origine al echipamentului.
24	Specifică dacă tasta este una extinsă, cum sunt ALT şi CTRL din partea dreaptă care apar pe tipul de tastatură extinsă cu 101-102 taste. Când bitul are valoarea 1, tasta este extinsă, iar dacă nu, tasta nu este extinsă.
25-28	Rezervate; nu se folosesc
29	Specifică un cod de context. Pentru *WM_KEYDOWN* valoarea este întotdeauna zero.
30	Specifică starea anterioară a tastei. Bitul este 1 dacă tasta este apăsată înainte ca mesajul să fie trimis şi nu este 1 dacă tasta nu este apăsată.
31	Specifică starea de tranziţie. Bitul nu este niciodată 1 pentru *WM_KEYDOWN*.

***Tabelul 1342.1** Valorile transmise de Windows în parametrul lParam cu mesajul WM_KEYDOWN.*

Dacă utilizatorul a acţionat tasta funcţională F10, funcţia *DefWindowProc* fixează un indicator intern. Când *DefWindowProc* primeşte mesajul *WM_KEYUP*, funcţia testează dacă indicatorul intern este 1 şi, dacă da, trimite un mesaj *WM_SYSCOMMAND* către fereastra principală. *DefWindowProc* fixează parametrul *wParam* al mesajului la valoarea *SC_KEYMENU*.

Datorită caracteristicii sistemului Windows de autorepetare, Windows poate să expedieze mai multe mesaje *WM_KEYDOWN* către aplicaţie, înainte de a transmite mesajul *WM_KEYUP*. Programul poate utiliza starea anterioară a tastei (bitul 30) pentru a determina dacă mesajul *WM_KEYDOWN* indică o primă tranziţie de apăsare sau o tranziţie repetată de apăsare.

Pentru tastaturile extinse cu 101 şi 102 taste, tastele extinse sunt ALT şi CTRL din partea dreaptă, tastele din secţiunea principală a tastaturii, tastele INS, DEL, HOME, END, PAGE UP, PAGE DOWN şi tastele de direcţie din partea dreaptă a tastaturii numerice şi tastele împărţit (/) şi ENTER în tastatura numerică. Şi alte tastaturi pot accepta bitul de tastă extinsă din parametrul *lKeyData*.

Pot apărea situaţii când programul dumneavoastră trebuie să preia un şir care reprezintă un nume de tastă. În secţiunea precedentă, aţi prelucrat taste cunoscute în instrucţiunea *Switch*. Interfaţa win32 API, însă, întreţine o listă a tastelor la care corespund toate codurile de taste funcţionale. Puteţi să utilizaţi funcţia *GetKeyNameText* pentru a prelua un şir cu numele tastei. Veţi invoca funcţia *GetKeyNameText* cu prototipul prezentat mai jos:

```
int GetKeyNameText(
    LONG lParam, // al doilea parametru al mesajului de la
                 // tastatura
    LPTSTR lpString, // adresa bufferului cu numele tastei
    int nSize // lungime maxima a sirului cu numele tastei
);
```

Parametrul *lpString* indică un buffer care primeşte numele tastei. Parametrul *nSize* specifică lungimea maximă, în caractere, a numelui tastei, inclusiv caracterul de terminare NULL (acest parametru trebuie să fie egal cu mărimea bufferului la care indică parametrul *lpString*), Parametrul *lParam* specifică al doilea parametru al mesajului de tastatură ce trebuie prelucrat (cum ar fi *WM_KEYDOWN*). Funcţia interpretează părţile din *lParam* listate în Tabelul 1342.2.

Biţi	Semnificaţie
16-23	Cod de scanare.
24	Indicator de tastă extinsă. Distinge anumite taste din tastatura extinsă.
25	Bitul „indiferent". Aplicaţia apelantă fixează acest bit pe 1 ca să indice că funcţia nu trebuie să facă distincţie între tastele ALT şi CTRL din dreapta şi din stânga.

***Tabelul 1342.2** Biţii interpretaţi de **GetKeyNameText** în parametrul **lParam**.*

Formatul şirului cu numele tastei depinde de schema tastaturii curente. Driverul de tastatură întreţine o listă cu nume sub forma unor şiruri de caractere ale tastelor cu nume mai lungi de un singur caracter. Numele este convertit de driverul de tastatură în acord cu schema tastaturii curente instalate. Numele unei taste caracter este caracterul însuşi. CD-ROM-ul care însoţeşte cartea de faţă cuprinde programul *Show_Keys.cpp* care afişează un nume de tastă de fiecare dată când utilizatorul a acţionat o tastă.

Aşa cum veţi vedea, în instrucţiunea *case* care tratează mesajul *WM_KEYDOWN* programul execută prelucrări semnificative legate de contextele de dispozitiv despre care veţi învăţa în secţiunile viitoare. Pentru scopul acestei secţiuni cel mai important apel este la funcţia *GetKeyNameText*, cum prezentăm mai jos:

```
GetKeyNameText (lParam, szName, 30);
```

Programul apelează *GetKeyNameText*, apoi afişează în fereastră valoarea din bufferul *szName*.

FIXAREA ŞI RESTABILIREA TIMPULUI DE DUBLU CLIC AL MOUSE-ULUI

C/C++ 1343

Aşa cum aţi învăţat, programele dumneavoastră vor executa prelucrări semnificative pentru captarea şi prelucrarea corectă a intrărilor de la utilizator. Una dintre cele mai obişnuite activităţi operate de utilizatori în programele realizate de dumneavoastră este executarea unui dublu clic de mouse pe o anumită opţiune. Operaţiunea de dublu clic este o acţionare de două ori a unui buton de mouse, a doua intervenind după o perioadă specifică de timp după prima. Vor exista ocazii când doriţi să controlaţi perioada de timp dinaintea celui de-al doilea clic, menţinând acţiunea ca dublu clic de mouse. În această situaţie, programul va utiliza funcţia *SetDoubleClickTime* pentru a controla viteza operaţiunii de dublu clic. Funcţia

SetDoubleClickTime fixează *timpul de dublu clic* al mouse-ului. Timpul de dublu clic este numărul maxim de milisecunde cuprins între primul și al doilea clic de mouse. Programul dumneavoastră va utiliza funcția *SetDoubleClickTime* cu prototipul descris mai jos:

```
BOOL SetDoubleClickTime(
  UINT uInterval // interval de dublu clic
);
```

Parametrul *uInterval* specifică numărul de milisecunde dintre primul și al doilea clic de mouse. Dacă parametrul este zero, Windows va utiliza valoarea implicită de dublu clic care este de 500 de milisecunde.

Observație: *Funcția* ***SetDoubleClickTime*** *modifică timpul de dublu clic pentru toate ferestrele din sistem. Dacă programele dumneavoastră schimbă timpul de dublu clic, ele vor trebui să restabilească timpul inițial la încheierea programului.*

La fel cum în programele dumneavoastră puteți să fixați intervalul de timp dintre două clicuri de mouse care să fie considerate dublu clic, în același mod puteți prelua timpul curent de dublu clic al mouse-ului utilizând funcția *GetDoubleClickTime*. Așa cum știți, operațiunea de dublu clic este o acționare de două ori a unui buton de mouse, a doua intervenind după o perioadă specifică de timp după prima. Timpul de dublu clic este numărul maxim de milisecunde care se pot scurge între primul și al doilea clic de mouse. Programul dumneavoastră va utiliza funcția *GetDoubleClickTime* cu prototipul descris mai jos:

```
UINT GetDoubleClickTime (void)
```

Dacă funcția reușește, valoarea returnată va specifica timpul de dublu clic al mouse-ului în milisecunde. Pentru a înțelege mai bine prelucrarea funcțiilor *GetDoubleClickTime* și *SetDoubleClickTime*, analizați programul *Get_Set_Dbl_Click.cpp*, conținut în CD-ROM-ul care însoțește cartea de față.

De fiecare dată când utilizatorul execută dublu clic cu mouse-ul, programul va afișa o casetă de mesaj care informează că programul a interceptat un dublu clic. Însă, când utilizatorul selectează articolul *Test!* din meniu, programul crește de fiecare dată timpul de dublu clic cu o zecime de secundă. Rețineți că programul, la încheierea sa, restabilește timpul inițial de dublu clic.

1344 INVERSAREA BUTOANELOR MOUSE-ULUI



Așa cum ați învățat în secțiunea 1343, vor exista ocazii când programele dumneavoastră vor modifica durata dintre două clicuri în operația de dublu clic de mouse. Asemănător, vor fi situații când utilizatorul va inversa butoanele mouse-ului – utilizând butonul din dreapta al mouse-ului pentru a executa operațiile butonului din stânga și utilizând butonul stânga al mouse-ului pentru a executa operațiile butonului din dreapta. Funcția *SwapMouseButton* inversează și/sau restabilește înțelesul de buton stânga sau buton dreapta ale mouse-ului. Windows pune la dispoziție inversarea butoanelor mouse-ului pentru acomodarea persoanelor care folosesc mouse-ul cu mâna stângă. De obicei, numai Control Panel apelează funcția *SwapMouseButton*, deși aplicația poate la rândul ei apela funcția. Dacă programele dumneavoastră trebuie să utilizeze funcția *SwapMouseButton*, o veți invoca după următorul prototip:

```
BOOL SwapMouseButton(
    BOOL fSwap // Inverseaza sau restabileste butoanele
);
```

Parametrul *fSwap* precizează dacă semnificația butoanelor mouse-ului sunt pe moment inversate sau restabilite. Dacă *fSwap* este True, butonul din stânga va genera mesaje de tip buton dreapta, iar butonul din dreapta va genera mesaje de tip buton stânga. Dacă *fSwap* este False, butoanele sunt stabilite la semnificația inițială. CD-ROM-ul care însoțește cartea de față cuprinde programul *Swap_B.cpp* care inversează starea butoanelor mouse-ului dreapta-stânga și invers, de fiecare dată când utilizatorul selectează opțiunea *Test!* din meniu.

Observație: *Mouse-ul este o resursă partajată și inversarea semnificației butoanelor afectează toate aplicațiile. Programele dumneavoastră trebuie să evite, pe cât posibil, inversarea butoanelor.*

DETERMINAREA ACȚIONĂRII UNEI TASTE DE CĂTRE UTILIZATOR

C/C++1345

Așa cum ați învățat, programele dumneavoastră pot executa prelucrări de fiecare dată când utilizatorul apasă o tastă sau execută clic cu butonul mouse-ului. Unele aplicații, însă, solicită utilizatorului executarea mai multor pași pentru selectarea unei opțiuni. De exemplu, un program poate răspunde diferit dacă utilizatorul apasă tasta funcțională F2 și apoi execută clic pe o opțiune sau dacă programul răspunde la un simplu clic pe opțiunea respectivă. Pentru executarea acestor procese asincrone (adică nu simultane), programele dumneavoastră pot utiliza funcția *GetAsyncKeyState*. Această funcție determină dacă o tastă este apăsată sau eliberată în momentul apelării funcției de către program și dacă utilizatorul a apăsat tasta după un apel anterior la *GetAsyncKeyState*. Veți implementa funcția după următorul prototip:

```
SHORT GetAsyncKeyState(
    int vKey // Cod de tasta virtuala
);
```

Parametrul *vKey* specifică unul dintre cele 256 de coduri posibile de taste virtuale, prezentate în tabelul 1340. Dacă funcția reușește, valoarea returnată va specifica dacă utilizatorul a apăsat o tastă în intervalul de la ultimul apel la *GetAsyncKeyState* și dacă tasta este pe moment apăsată sau nu. Dacă bitul cel mai semnificativ este 1, tasta este apăsată, iar dacă bitul mai puțin semnificativ este 1, utilizatorul a apăsat tasta după un apel anterior la funcția *GetAsyncKeyState*. Valoarea returnată va fi 0 dacă fereastra dintr-un alt fir de execuție sau proces deține, pe moment, focusul intrărilor de la tastatură.

Observație: *Sub Windows 95, puteți utiliza constantele de coduri de taste virtuale* ***VK_SHIFT***, ***VK_CTRL*** *și* ***VK_MENU*** *ca valori pentru parametrul* ***vKey***. *Aceasta redă starea tastelor SHIFT, CTRL sau ALT fără diferențierea între stânga și dreapta.*

Observație: *Sub Windows NT puteți utiliza următoarele constante de coduri de taste virtuale ca valori pentru* ***vKey***, *pentru a distinge între instanțele stânga și dreapta ale acestor taste:*

VK_LSHIFT *VK_RSHIFT*
VK_LCONTROL *VK_RCONTROL*
VK_LMENU *VK_RMENU*

Constantele de diferențiere stânga-dreapta sunt disponibile numai când apelați funcțiile ***GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState*** *și* ***MapVirtualKey.***

Funcția *GetAsyncKeyState* operează cu butoanele mouse-ului. Ea testează, însă, starea fizică a butoanelor mouse-ului și nu starea logică a butoanelor la care butoanele fizice sunt mapate. De exemplu, apelul la *GetAsyncKeyState(VK_LBUTTON)* va returna întotdeauna starea butonului fizic din stânga pe care sistemul l-a mapat ca buton logic din stânga sau din dreapta al mouse-ului. Puteți determina maparea curentă de sistem a butoanelor fizice la butoanele logice ale mouse-ului apelând funcția *GetSystemMetrics(SM_SWAPBUTTON)*. Ea returnează True dacă sistemul de operare a inversat anterior butoanele mouse-ului și False în situația contrară. Pentru a înțelege mai bine prelucrările efectuate de funcția *GetAsyncKeyState*, să analizăm programul *Get_Async_Keys.cpp* cuprins în CD-ROM-ul care însoțește cartea de față. Programul *Get_Async_Keys.cpp* returnează starea curentă a tastelor SHIFT când utilizatorul selectează articolul *Test!* din meniu.

1346 BARELE DE DERULARE C/C++

Așa cum ați învățat, fereastra este o aplicație care poate afișa multe obiecte. Din timp în timp fereastra poate afișa obiecte cuprinzând date, cum ar fi un document sau o imagine grafică mai largă decât zona client a ferestrei. Când fereastra deține o bară de derulare, utilizatorul poate derula obiectul în cadrul zonei client pentru a putea vedea în întregime documentul sau imaginea grafică. Aplicațiile trebuie să atașeze bare de derulare la ferestre de fiecare dată când conținutul zonei client se extinde peste dimensiunea zonei client a ferestrei. O bară de derulare standard are trei componente: câte o săgeată de fiecare parte, lungimea sau înălțimea barei și un buton de derulare – o casetă de dimensiuni variabile (în Windows 95/NT, fixe în Windows 3x) pe care utilizatorul o deplasează înainte și înapoi de-a lungul barei. Figura 1346 prezintă ca exemplu un program cu bare de derulare pe marginile orizontală și verticală, precum și componentele lor.

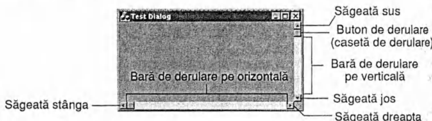


Figura 1346 *O fereastră cu două bare interioare de derulare și componentele lor.*

DIFERITELE TIPURI DE BARE DE DERULARE

C/C++1347

Aşa cum aţi învăţat în secţiunea 1346, orice bară de derulare standard, indiferent de poziţie sau direcţie, are anumite caracteristici comune cu alte bare de derulare standard. Aşa cum aţi văzut în Figura 1346, există cel puţin două tipuri de bare de derulare. De fapt, există două categorii de bare de derulare. Prima categorie cuprinde *barele de derulare din zona client*. Acest tip de bare se pot extinde orizontal sau vertical şi pot avea aspectul unei bare de derulare standard sau a unei bare de deplasare. A doua categorie cuprinde *barele de derulare din zona neclient*. Windows ataşează barele de derulare din zona neclient la chenarul ferestrei, imediat ce faceţi vizibilă bara de derulare.

Aşa cum ştiţi, puteţi crea în aplicaţii bare de derulare în zona neclient în cadrul structurii *WNDCLASS* pe care programul o transmite funcţiei *CreateWindow*. Când ataşaţi o bară de derulare la o margine a ferestrei, în mod automat Windows scade lăţimea barei din zona client a ferestrei, în aşa fel încât trasarea în zona client să nu se facă peste bara de derulare. Programul poate, de asemenea, invoca funcţia *ShowScrollBar* pentru a ataşa bare de derulare la marginea interioară a ferestrelor. Veţi învăţa mai mult despre *ShowScrollBar* în secţiunea 1348.

În plus faţă de ataşarea barelor de derulare la marginea interioară a ferestrelor, sistemul de operare va ataşa automat barele de derularea la casetele listă şi casetele combinate, când lista de articole depăşeşte mărimea ferestrei listă. Programul poate, de asemenea, ataşa bare de derulare la casete de editare. Casetele de editare cu o singură linie, însă, acceptă numai bara de derulare orizontală, pe când cele multilinie, acceptă barele de derulare atât pe orizontală cât şi pe verticală.

UTILIZAREA FUNCŢIEI SHOWSCROLLBAR

C/C++1348

Aşa cum aţi învăţat în secţiunea 1347, programul dumneavoastră poate ataşa bare de derulare la ferestre, atât înainte de crearea lor, cât şi după aceea. Pentru a ataşa o bară de derulare la o fereastră produsă anterior, programul dumneavoastră va utiliza funcţia *ShowScrollBar*. Această funcţie arată sau ascunde barele de derulare specificate. Funcţia *ShowScrollBar* va fi utilizată în program cu următorul prototip:

```
BOOL ShowScrollBar(
    HWND hWnd,  // identificatorul ferestrei cu bara de derulare
    int wBar,   // indicator pentru bara de derulare
    BOOL bShow  // indicator de valabilitate bara
);
```

Parametrul *hWnd* identifică un control bară de derulare sau o fereastră cu o bară de derulare standard, în funcţie de valoarea parametrului *wBar*. Parametrul *bShow* specifică dacă bara de derulare este sau nu vizibilă. Dacă parametrul este True (adevărat), Windows va afişa bara de derulare, iar dacă valoarea este False, va ascunde bara de derulare. Parametrul *wBar* specifică barele care vor fi arătate sau ascunse şi poate lua una din valorile listate în Tabelul 1348.1.

Valoare	Semnificație
SB_BOTH	Arată sau ascunde barele de derulare standard orizontale și verticale.
SB_CTL	Arată sau ascunde un control bară de derulare. Parametrul *hWnd* trebuie să fie identificatorul controlului bară de derulare.
SB_HORZ	Arată sau ascunde barele de derulare standard orizontale ale ferestrei.
SB_VERT	Arată sau ascunde barele de derulare standard verticale ale ferestrei.

Tabelul 1348.1 *Valorile acceptate de parametrul* ***wBar***.

Observație: *Nu trebuie să apelați funcția* ***ShowScrollBar*** *pentru a ascunde o bară de derulare în timpul tratării unui mesaj tip bară de derulare.*

După ce faceți vizibile barele de derulare, puteți să controlați modul în care poate utilizatorul manipula barele respective. Unul din cele mai simple mijloace de control pe care le veți aplica este de a activa sau dezactiva una sau ambele săgeți din gara de derulare. În acest caz, programul va apela funcția *EnableScrollBar* care activează sau dezactivează una sau ambele săgeți ale barei de derulare. Funcția *EnableScrollBar* se va implementa în modul arătat mai jos:

```
BOOL EnableScrollBar(
    HWND hWnd,          // identificatorul ferestrei cu bara de
                        // derulare
    UINT wSBflags,      // indicator cu tipul barei de derulare
    UINT wArrows    // indicator pt sagetile barei de derulare
);
```

Parametrul *hWnd* identifică controlul fereastră sau bara de derulare, în funcție de parametrul *wSBflags*. Parametrul *wSBflags* specifică tipul barei de derulare și ia una din valorile listate în Tabelul 1348.1.

Parametrul *wArrows* specifică dacă săgețile barei de derulare sunt activate sau dezactivate și precizează ce săgeată va fi activată sau dezactivată. Parametrul *wArrows* poate lua una din valorile listate în Tabelul 1348.2.

Valoare	Semnificație
ESB_DISABLE_BOTH	Dezactivează ambele săgeți ale barei de derulare
ESB_DISABLE_DOWN	Dezactivează săgeata de jos a barei de derulare
ESB_DISABLE_LEFT	Dezactivează săgeata din stânga a barei de derulare
ESB_DISABLE_LTUP	Dezactivează săgeata din stânga a barei de derulare orizontale sau săgeata de sus a unei bare de derulare verticale
ESB_DISABLE_RIGHT	Dezactivează săgeata din dreapta a unei bare de derulare orizontale
ESB_DISABLE_RTDN	Dezactivează săgeata din dreapta a barei de derulare orizontale sau săgeata de jos a unei bare de derulare verticale
ESB_DISABLE_UP	Dezactivează săgeata de sus a unei bare de derulare verticale
ESB_ENABLE_BOTH	Activează ambele săgeți ale unei bare de derulare

Tabelul 1348.2 *Valorile acceptate de parametrul* ***wArrows***.

Doar cu aceste două funcții, programele pot executa cea mai mare parte din operațiile necesare cu bara de derulare. CD-ROM-ul atașat conține programul *Scroll.cpp*, care folosește ambele funcții *ShowScrollBars* și *EnableScrollBars* pentru a controla bara de derulare în program.

POZIȚIA ȘI INTERVALUL UNEI BARE DE DERULARE

C/C++1349

Așa cum ați învățat în secțiunea 1348, adăugarea unei bare de derulare la o fereastră este un proces simplu. Când creați o bară de derulare, intervalul implicit al valorilor reprezentate de cele două capete ale controlului este de 0 la 100. În majoritatea cazurilor, aplicațiile vor schimba intervalul pentru a reflecta dimensiunea documentului sau imaginii. Poziția butonului de derulare este acea valoare din interval în care este poziționat butonul. De exemplu, dacă un interval este de 0 la 100 și poziția butonului de derulare este la 50, butonul va apărea la jumătatea distanței dintre capetele controlului.

Puteți, de asemenea, fixa *dimensiunea paginii* pentru controlul bară de derulare. Dimensiunea paginii reprezintă numărul de incrementări din cadrul intervalului de derulare pe care pagina îl poate afișa o dată. De exemplu, dacă intervalul este de 0 la 100 și dimensiunea paginii este fixată la 50, pagina poate afișa o dată jumătate din intervalul controlului. Programul dumneavoastră poate utiliza funcțiile *SetScrollInfo* și *GetScrollInfo* pentru a fixa și a prelua intervalul barei de derulare, poziția butonului de derulare și dimensiunea paginii.

MESAJELE BAREI DE DERULARE

C/C++1350

În secțiunile precedente, ați învățat câteva lucruri de bază despre barele de derulare. Pe măsură ce utilizatorul folosește bara de derulare aceasta generează mesaje către aplicație, la fel ca mouse-ul sau tastatura. Când utilizatorul execută clic cu mouse-ul pe bara de derulare, Windows trimite un mesaj *WM_HSCROLL* sau *WM_VSCROLL* către aplicație, în funcție de orientarea orizontală sau verticală a barei. Cuvântul mai puțin semnificativ al parametrului *lParam* comunică funcției de prelucrare poziția mouse-ului în momentul când utilizatorul a executat clicul. În funcție de localizarea mouse-ului la momentul clicului, Windows va trimite unul din zece mesaje către program. Figura 1350 prezintă poziția posibilă a clicului de mouse și mesajul de fereastră corespunzător.

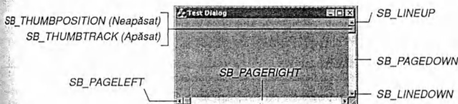


Figura 1350 *Câteva din mesajele generate de Windows la executarea unui clic de mouse pe barele de derulare.*

Când utilizatorul eliberează butonul mouse-ului după executarea unui clic oriunde pe o bară de derulare, dacă utilizatorul nu acționează cu mouse-ul butonul de derulare, Windows va

trimite către aplicație mesajul *SB_ENDSCROLL*. În acest caz, când utilizatorul va elibera butonul mouse-ului, Windows va genera mesajul *SB_THUMBPOSITION*.

1351 OBȚINEREA CONFIGURAȚIEI CURENTE A BAREI DE DERULARE



Așa cum ați învățat, în programul dumneavoastră barele de derulare încep cu anumite configurații implicite, când le adăugați la ferestre. E posibil, însă, ca programul dumneavoastră să modifice configurația inițială pe parcursul prelucrării programului. Deseori programul va trebui să execute prelucrări speciale bazate pe configurarea barelor de derulare. Pentru a determina caracteristicile unei bare de derulare, programul dumneavoastră poate folosi funcția *GetScrollInfo*. Funcția *GetScrollInfo* preia parametrii unei bare de derulare, inclusiv poziția minimă și maximă de derulare, dimensiunea paginii și poziția butonului de derulare. Veți utiliza funcția *GetScrollInfo* cu prototipul prezentat mai jos:

```
BOOL GetScrollInfo(
   HWND hwnd, // identificator al ferestrei cu bara de derulare
   int fnBar, // indicator pentru bara de derulare
   LPSCROLLINFO lpsi // pointer la structura cu parametrii de
                     // derulare
);
```

Așa cum puteți vedea funcția *GetScrollInfo* acceptă trei parametri. Parametrul *lpsi* despre care veți învăța mai târziu în această secțiune, returnează o valoare de tip *SCROLLINFO*, o structură definită de sistem ai cărei membri sunt:

```
typedef struct tagSCROLLINFO
  {
    UINT cbSize;
    UINT fMask;
    int nMin;
    int nMax;
    UINT nPage;
    int nPos;
    int nTrackPos;
  } SCROLLINFO;
```

Atât *GetScrollInfo* cât și *SetScrollInfo* utilizează structura *SCROLLINFO*. Tabelul 1351.1 prezintă în detaliu membrii structurii și utilizarea lor.

Membru	Descriere
cbSize	Specifică dimensiunea în octeți a structurii
fMask	Specifică parametrii barei de derulare pentru preluare sau fixare. Acest membru poate fi o combinație de valori prezentate în Tabelul 1351.2.
NMin	Specifică poziția minimă de derulare.
Nmax	Specifică poziția maximă de derulare.

Membru	Descriere
Npage	Specifică dimensiunea paginii. Bara de derulare utilizează această valoare pentru a determina dimensiunea corespunzătoare a casetei de derulare proporționale.
Npos	Specifică poziția casetei de derulare.
NtrackPos	Specifică poziția imediată a unei casete de derulare pe care utilizatorul o deplasează. Aplicația poate prelua această valoare în timpul tratării mesajului de notificare *SB_THUMBTRACK*. Aplicația nu poate stabili imediat poziția de derulare; funcția *SetScrollInfo* ignoră acest membru.

Tabelul 1351.1 *Membrii Structurii **SCROLLINFO**.*

Așa cum ați aflat din Tabelul 1351.1, membrul *fMask* poate avea una din valorile predefinite, prezentate în Tabelul 1351.2.

Membru	Descriere
SIF_ALL	Combinație de *SIF_PAGE*, *SIF_POS* și *SIF_RANGE*.
SIF_DISABLENOSCROLL	Această valoare este importantă numai când stabilim parametrii barei de derulare. Dacă noii parametri ai barei de derulare nu mai fac necesară bara, trebuie dezactivată bara și nu ștearsă.
SIF_PAGE	Membrul *nPage* conține dimensiunea paginii unei bare de derulare proporționale
SIF_POS	Parametrul *nPos* conține poziția casetei de derulare
SIF_RANGE	Membrii *nMin* și *nMax* conțin valorile minime și maxime ale intervalului de derulare

Tabelul 1351.2 *Valorile predefinite ale membrului **fMask**.*

În cadrul funcției *GetScrollInfo*, parametrul *hWnd* identifică un control bară de derulare sau o fereastră cu o bară de derulare standard, în funcție de valoarea parametrului *fnBar*. Parametrul *fnBar* specifică tipul barei de derulare unde se regăsesc parametrii. Parametrul *fnBar* poate lua una din valorile prezentate în Tabelul 1351.3.

Membru	Descriere
SB_CTL	Preia parametrii controlului bară de derulare. Parametrul *hwnd* trebuie să fie identificatorul controlului bară de derulare.
SB_HORZ	Preia parametrii unei bare de derulare standard orizontale.
SB_VERT	Preia parametrii unei bare de derulare standard verticale.

Tabelul 1351.3 *Valorile acceptate de parametrul **fnBar**.*

În sfârșit, parametrul *lpsi* indică structura *SCROLLINFO*, al cărei membru *fMask*, de la intrarea în funcție, specifică parametrii barei de derulare care urmează a fi preluați. Înainte de returnare, funcția copiază parametrii specificați în membrii corespunzători ai structurii. Membrul *fMask* poate fi o combinație a valorilor prezentate în Tabelul 1351.4 (rețineți că funcția *GetScrollInfo* acceptă numai anumite valori ale membrului *fMask*).

Membru	Descriere
SIF_PAGE	Copiază pagina de derulare în membrul *nPage* al structurii *SCROLLINFO* indicat de parametrul *lpsi*
SIF_POS	Copiază poziția de derulare în membrul *nPos* al structurii *SCROLLINFO* indicat de parametrul *lpsi*
SIF_RANGE	Copiază intervalul de derulare în membrii *nMin* și *nMax* ai structurii *SCROLLINFO* indicați de parametrul *lpsi*

Tabelul 1351.4 *Valorile acceptate ale membrului **fMask** al structurii **SCROLLINFO**.*

Funcția *GetScrollInfo* permite aplicațiilor să folosească poziții de derulare pe 32 de biți. Deși mesajele care indică pozițiile de derulare, *WM_HSCROLL* și *WM_VSCROLL*, pun la dispoziție date de poziționare numai pe 16 biți, funcțiile *SetScrollInfo* și *GetScrollInfo* oferă aceste date pe 32 de biți. În consecință, aplicațiile pot apela *GetScrollInfo* pentru a obține date despre poziția barei de derulare pe 32 de biți, în timp ce tratează mesajele, *WM_HSCROLL* sau *WM_VSCROLL*.

Limitarea accesului la poziția barei de derulare se aplică procesului de derulare în timp real a conținutului ferestrei. Aplicațiile implementează derularea în timp real prin prelucrarea mesajelor *WM_HSCROLL* sau *WM_VSCROLL* care poartă valoarea de notificare *SB_THUMBTRACK*, cu poziția casetei de derulare deplasate de utilizator. Din păcate, nu există nici o funcție de preluare a poziției casetei de derulare pe 32 de biți în timp ce utilizatorul deplasează caseta (butonul) de derulare. Deoarece *GetScrollInfo* pune la dispoziție numai poziția statică, aplicațiile pot obține poziția pe 32 de biți numai înainte sau după operațiile de derulare.

1352 DERULAREA CONȚINUTULUI FERESTREI



Așa cum vă imaginați, cea mai importantă funcție executată de barele de derulare este derularea conținutului unei ferestre. Veți controla derularea ferestrei cu funcția *ScrollWindowEx* care derulează conținutul unei ferestre specificate din zona client. În programele dumneavoastră, veți utiliza funcția *ScrollWindowEx* după prototipul de mai jos:

```
int ScrollWindowEx(
        HWND hWnd,      // identificator al ferestrei de derulat
        int dx,         // volum derulare orizontala
        int dy,         // volum derulare verticala
        CONST RECT *prcScroll, // adresa structurii cu
                               // dreptunghiul de derulare
        CONST RECT *prcClip,   // adresa structurii cu
                               // dreptunghiul pentru decupare
        HRGN hrgnUpdate,       // identificator al regiunii
                               // actualizate
        LPRECT prcUpdate,      // adresa structurii cu
                               // dreptunghiul actualizat
        UINT flags             // indicatoare de derulare
);
```

Așa cum puteți vedea, funcția *ScrollWindowEx* are opt parametri, majoritatea previzibili. Tabelul 1352.1 prezintă parametrii acestei funcții.

Parametru	Descriere
hWnd	Identifică fereastra cu zona client de derulare.
dx	Specifică volumul, în unități de dispozitiv, al derulării orizontale. Acest parametru trebuie să fie o valoare negativă pentru a derula la stânga.
dy	Specifică volumul, în unități de dispozitiv, al derulării verticale. Acest parametru trebuie să fie o valoare negativă pentru a derula în sus.
prcScroll	Indică o structură care specifică porțiunea zonei client de derulare. Dacă acest parametru este NULL, întreaga zonă client este derulată.
prcClip	Indică o structură care conține coordonatele dreptunghiului de decupare. Numai biții de dispozitiv din dreptunghiul de decupare sunt afectați. Biții derulați dinspre exteriorul dreptunghiului spre interior sunt desenați; biții derulați dinspre interiorul dreptunghiului spre exterior nu sunt desenați.
hrgnUpdate	Identifică regiunea modificată de *ScrollWindowEx* pentru a reține regiunea invalidată prin derulare. Acest parametru poate să fie NULL.
prcUpdate	Indică o structură RECT care primește marginile dreptunghiului invalidat prin derulare. Acest parametru poate să fie NULL.
flags	Specifică indicatoarele care controlează derularea. Acest parametru trebuie să ia una dintre valorile prezentate în Tabelul 1352.2.

Tabelul 1352.1 Parametrii funcției ***ScrollWindowEx***.

Așa cum indică Tabelul 1352.1 parametrul *flags* poate lua o valoare dintr-un set de valori predefinite. Tabelul 1352.2 prezintă valorile acceptate de parametrul *flags*.

Valoare	Semnificație
SW_ERASE	Șterge noua regiune invalidată prin trimiterea mesajului *WM_ ERASEBKGND* către fereastră când este specificat indicatorul *SW_INVALIDATE*.
SW_INVALIDATE	Invalidează regiunea identificată de parametrul hrgnUpdate după derulare.
SW_SCROLLCHILDREN	Derulează toate ferestrele copil care intersectează dreptunghiul spre care indică parametrul *prcScroll*. Ferestrele copil sunt derulate cu numărul de pixeli specificați de parametrii *dx* și *dy*. Windows trimite mesajul *WM_MOVE* către toate ferestrele copil care intersectează dreptunghiul *prcScroll* chiar dacă ele nu se mișcă.

Tabelul 1352.2 Valorile pe care le ia parametrul flags.

Dacă funcția reușește, valoarea returnată este *SIMPLEREGION* (regiune dreptunghiulară invalidată), *COMPLEXREGION* (regiune ne-dreptunghiulară invalidată; dreptunghiuri suprapuse) sau *NULLREGION* (nici o regiune invalidată). Dacă funcția eșuează, valoarea returnată va fi *ERROR*.

Dacă apelul funcției nu specifică indicatoarele *SW_INVALIDATE* și *SW_ERASE*, funcția *ScrollWindowEx* nu invalidează zona de derulare. Dacă unul din aceste indicatoare este 1, *ScrollWindowEx* invalidează această zonă. Windows nu va actualiza zona până când aplicația nu apelează funcțiile *UpdateWindow*, *RedrawWindow* (care specifică indicatorul *RDW_UPDATENOW* sau *RDW_ERASENOW*) sau preia mesajul *WM_PAINT* din coada de mesaje a aplicației.

Dacă fereastra are stilul *WS_CLIPCHILDREN*, zonele returnate specificate de *hrgnUpdate* și *prcUpdate* reprezintă zona totală a ferestrei derulate care trebuie actualizată, inclusiv orice zonă din ferestrele copil ce necesită a fi actualizate. Dacă este precizat indicatorul *SW_SCROLLCHILDREN*, Windows nu va actualiza corespunzător ecranul în cazul în care este derulată secțiunea unei ferestre copil. Partea din fereastra copil care se află în afara dreptunghiului sursă nu este ștearsă și nu este corespunzător redesenată la noua sa destinație. Pentru a muta ferestre copil care nu se află complet în dreptunghiul specificat de *prcScroll*, utilizați funcția *DeferWindowPos*. Cursorul este repoziționat dacă indicatorul *SW_SCROLLCHILDREN* este 1 și dreptunghiul care lipsește intersectează dreptunghiul de derulat.

Windows determină toate coordonatele de intrare și de ieșire (pentru *prcScroll, prcClip, prcUpdate și hrgnUpdate*) drept coordonate client, indiferent dacă fereastra prezintă stilurile clasă *CS_OWNDC* sau *CS_CLASSDC*. Utilizați funcțiile *LPtoDP* și *DPtoLP* pentru a converti din și în coordonate logice, dacă este necesar.

Pentru a înțelege mai bine procesul executat de funcția *ScrollWindowEx*, să analizăm programul *Scroll_Window.cpp* conținut în CD-ROM-ul care însoțește cartea de față. Programul *Scroll_Window.cpp* permite utilizatorului să adauge câte o linie în fereastra programului. După ce numărul de linii depășește volumul spațiului din fereastră, bara de derulare devine activă. Când utilizatorul execută clic cu mouse-ul pe săgeata de sus sau de jos, butonul de derulare se mișcă cu o linie în direcția corespunzătoare.

1353 MESAJUL WM_SIZE



În secțiunea 1352 ați învățat despre programul *Scroll_Window.cpp*, care permite utilizatorului să selecteze bara de derulare pentru deplasare în sus și în jos în fereastră. Programul doar activează bara de derulare când conținutul ferestrei devine prea lung pentru a se desfășura în fereastra cu dimensiunile curente. Dacă încercați programul *Scroll_Window* și lărgiți fereastra, veți observa că bara de derulare devine inactivă din nou după ce lărgiți fereastra suficient pentru a afișa textul. Programul *Scroll_Window* captează mesajul de sistem *WM_SIZE* pentru a ajusta bara de derulare, după ce ați redimensionat fereastra.

```
      // De fiecare data cand fereastra este dimensionata, se
      // recalculeaza numarul de linii pe  are le poate afisa
           // zona client si se configureaza bara de
           // derulare corespunzator
case WM_SIZE :
        {
             RECT rect;
             GetClientRect( hWnd, &rect );
             nDspLines = rect.bottom / 20;
             if ( nDspLines < nNumItems )
             {
              SCROLLINFO si;
              si.cbSize = sizeof( SCROLLINFO );
              si.fMask = SIF_POS | SIF_RANGE | SIF_PAGE;
              si.nMin = 0;
              si.nMax = nNumItems-1;
```

```
            si.nPage = nDspLines;
            si.nPos = nCurPos;
            EnableScrollBar( hWnd, SB_VERT, ESB_ENABLE_BOTH );
            SetScrollInfo( hWnd, SB_VERT, &si, TRUE );
          }
          else
            EnableScrollBar( hWnd, SB_VERT, ESB_DISABLE_BOTH );
        }
        break;
```

În cazul programului *Scroll_Window.cpp*, instrucţiunea *case WM_SIZE* testează numărul de linii pe care îl poate afişa fereastra în raport cu numărul de linii afişate curent. Dacă numărul de linii posibile este mai mare decât numărul de linii afişate, programul va dezactiva bara de derulare. Altfel, el va modifica dimensiunea butonului de derulare (când este necesar) şi va afişa bara de derulare recent dimensionată.

Programul dumneavoastră va utiliza mesajul *WM_SIZE* pentru a testa dimensiunea curentă a ferestrei şi a înlocui orice element din fereastră care depinde de dimensiunea sa. De exemplu, dacă aveţi o casetă de editare într-o fereastră care afişează un titlu şi utilizatorul redimensionează fereastra, puteţi să redimensionaţi caseta de editare împreună cu fereastra. Acţiunea executată de program atunci când primeşte mesajul *WM_SIZE* va fi diferit în funcţie de tipul redimensionării executate de utilizator. Mesajul *WM_SIZE* transmite o constantă care reprezintă tipul de redimensionare în cadrul parametrului *wParam*. Tabelul 1353 prezintă valorile posibile ale parametrului *wParam*.

Valoare	Semnificaţie
SIZE_MAXHIDE	Mesajul este trimis la toate ferestrele derulante, când o altă fereastră este maximizată de utilizator.
SIZE_MAXIMIZED	Fereastra a fost maximizată de utilizator.
SIZE_MAXSHOW	Mesajul este trimis de Windows la toate ferestrele derulante când una din celelalte ferestre a fost readusă la forma iniţială de utilizator.
SIZE_MINIMIZED	Fereastra a fost minimizată de utilizator.
SIZE_RESTORED	Fereastra a fost redimensionată de utilizator, dar nu se aplică nici valoarea *SIZE_MINIMIZED*, nici valoarea *SIZE_MAXIMIZED*.

Tabelul 1353 *Valorile acceptate de parametrul* ***wParam*** *într-un mesaj* ***WM_SIZE***.

În plus, mesajul *WM_SIZE* transmite informaţii despre noua fereastră în parametrul *lParam*. Cuvântul mai puţin semnificativ al parametrului *lParam* specifică noua dimensiune a zonei client, iar cuvântul cel mai semnificativ specifică noua înălţime a zonei.

MESAJUL WM_PAINT

C/C++1354

Aşa cum aţi învăţat în secţiunea 1354, aplicaţia dumneavoastră va primi mesajul *WM_SIZE* de fiecare dată când utilizatorul redimensionează fereastra aplicaţiei. Windows va trimite către aplicaţie şi mesajul *WM_PAINT* care urmează lui *WM_SIZE*, de fiecare dată când utilizatorul redimensionează fereastra aplicaţiei. De fapt, aplicaţia primeşte mesajul *WM_PAINT* când Windows sau o altă aplicaţie cere să deseneze o porţiune din fereastra aplicaţiei. Windows trimite mesajul *WM_PAINT* când programul apelează funcţiile *UpdateWindow* sau

RedrawWindow, precum și atunci când trimite funcția *DispatchMessage* în cazul în care aplicația utilizează *GetMessage* sau *PeekMessage* pentru a obține un mesaj *WM_PAINT*.

Parametrul *wParam* al mesajului *WM_PAINT* conține valoarea *hdc* care identifică un context de dispozitiv în care Windows va desena. Dacă parametrul *wParam* este NULL, aplicația trebuie să utilizeze valoarea implicită a contextului de dispozitiv (în loc de a crea un context de dispozitiv privat). Parametrul este utilizat de unele controale uzuale pentru a desena în contextul de dispozitiv, altul decât cel implicit. Alte ferestre pot ignora fără probleme acest parametru. Veți învăța mai mult despre contextele de dispozitiv în lecțiile următoare.

Funcția *DefWindowProc* validează regiunea actualizată. Funcția poate, de asemenea, trimite mesajul *WM_NCPAINT* către procedura fereastră, dacă chenarul ferestrei trebuie desenat și trimite mesajul *WM_ERASEBKGND*, dacă fundalul ferestrei trebuie șters.

Sistemul trimite acest mesaj când nu mai sunt alte mesaje în coada de mesaje a aplicației. Funcția *DispatchMessage* determină unde trebuie trimis mesajul; funcția *GetMessage* determină mesajul ce trebuie distribuit. *GetMessage* returnează mesajul *WM_PAINT* când nu mai sunt alte mesaje în coada de mesaje a aplicație, iar *DispatchMessage* trimite mesajul la procedura fereastră corespunzătoare.

Fereastra poate primi mesaje de desenare interne, ca rezultat al apelării lui *RedrawWindow* cu indicatorul *RDW_INTERNALPAINT* activat. În acest caz, fereastra nu poate avea o regiune de actualizare. Aplicația trebuie să apeleze funcția *GetUpdateRect* pentru a determina dacă fereastra are o regiune de actualizare. Dacă funcția *GetUpdateRect* returnează zero, aplicația nu trebuie să apeleze funcțiile *BeginPaint* și *EndPaint*. Dacă aplicația nu apelează funcția *GetUpdateRect*, e posibil ca unele porțiuni din fereastră să nu se actualizeze deloc corect.

Aplicația trebuie să testeze orice necesitate de desenare internă prin examinarea fiecărui mesaj *WM_PAINT* în structura internă de date, pentru că mesajul *WM_PAINT* a fost probabil cauzat de o regiune de actualizare cu o valoare neNULL sau de apelul la funcția *RedrawWindow* cu indicatorul *RDW_INTERNALPAINT* activ.

Windows trimite un mesaj intern *WM_PAINT* o singură dată. După ce mesajul intern *WM_PAINT* este returnat de *GetMessage* sau *UpdateWindow* transmite *PeekMessage* unei ferestre, Windows nu expediază sau trimite mai departe mesajul *WM_PAINT* până când fereastra nu este invalidată sau până când nu este din nou apelată funcția *RedrawWindow* cu indicatorul *RDW_INTERNALPAINT* activ.

Pentru unele controale obișnuite, prelucrarea implicită a mesajului *WM_PAINT* testează parametrul *wParam*. Dacă *wParam* este neNULL, controlul presupune că valoarea este un *identificator de context de dispozitiv* și va desena folosind contextul dispozitiv. Pentru a înțelege mai bine prelucrarea făcută de program la primirea mesajului *WM_PAINT*, consultați programul *Scroll_Window.cpp* prezentat în secțiunea 1353.

1355 ALTE MESAJE PENTRU BARA DE DERULARE CAPTATE DE PROGRAM



Așa cum ați învățat, barele de derulare generează mesaje specifice de derulare, în funcție de zona barei de derulare în care utilizatorul execută clic cu mouse-ul. În plus față de mesajele *WM_SIZE* și *WM_PAINT* despre care ați învățat în secțiunile 1353 și 1354, programele dumneavoastră trebuie, de asemenea, să capteze mesajele *WM_VSCROLL* și *WM_HSCROLL*.

Windows trimite mesajul *WM_VSCROLL* către o fereastră când se declanşează un eveniment de derulare în bara de derulare standard verticală. Windows trimite, de asemenea, acest mesaj proprietarului controlului bară de derulare verticală, când un eveniment de derulare se declanşează în control. Când programul primeşte un mesaj *WM_VSCROLL*, cuvântul mai puţin semnificativ al parametrului *wParam* specifică o valoare care precizează solicitarea de derulare a utilizatorului. Tabelul 1355.1 listează valorile acceptate de cuvântul mai puţin semnificativ.

Valoare	Semnificaţie
SB_BOTTOM	Derulează în dreapta jos.
SB_ENDSCROLL	Încheie derularea.
SB_LINEDOWN	Derulează cu o linie mai jos.
SB_LINEUP	Derulează cu o linie mai sus.
SB_PAGEDOWN	Derulează cu o pagină mai jos.
SB_PAGEUP	Derulează cu o pagină mai sus.
SB_THUMBPOSITION	Derulează la poziţia absolută (un deplasament numeric începând exact de la marginea ferestrei, cum ar fi 12 linii în jos faţă de marginea de sus). Poziţia curentă este dată de parametrul *nPos*.
SB_THUMBTRACK	Deplasează caseta de derulare la poziţia specificată. Poziţia curentă este precizată de parametrul *nPos*.
SB_TOP	Derulează în stânga sus.

Tabelul 1355.1 *Valorile posibile pentru cuvântul mai puţin semnificativ al parametrului wParam.*

Pe lângă valoarea pe care Windows o transmite în cuvântul mai puţin semnificativ al parametrului *wParam*, programul trebuie de asemenea să testeze cuvântul mai semnificativ. Cuvântul mai semnificativ specifică poziţia curentă a casetei de derulare, dacă parametrul *nScrollCode* ia valorile *SB_THUMBPOSITION* sau *SB_THUMBTRACK;* altfel, cuvântul mai semnificativ conţine o valoare inutilizabilă. În sfârşit, parametrul *lParam* conţine identificatorul controlului bară de derulare. Dacă bara de derulare nu trimite mesajul, *lParam* este NULL.

Aplicaţiile care răspund la deplasarea de către utilizator a casetei de derulare, de regulă, utilizează mesajul de notificare *SB_THUMBTRACK*. Dacă aplicaţia derulează conţinutul unei ferestre, ea trebuie de asemenea să restabilească poziţia casetei de derulare prin utilizarea funcţiei *SetScrollPos*.

Windows trimite mesajul *WM_HSCROLL* către fereastră când este declanşat un eveniment în bara de derulare standard orizontală. Windows trimite, de asemenea, acest mesaj către posesorul barei de derulare standard orizontală, când se declanşează un eveniment de derulare în cadrul controlului. La fel cum mesajul *WM_SCROLL* cuprinde informaţii suplimentare în parametrii *wParam şi lParam*, în acelaşi mod procedează şi mesajul *WM_HSCROLL*.

Cu *WM_HSCROLL*, cuvântul mai puţin semnificativ al parametrului *wParam* specifică o valoare care precizează solicitarea de derulare a utilizatorului. Tabelul 1355.2 listează valorile acceptate de cuvântul mai puţin semnificativ al parametrului *wParam*.

Valoare	Semnificație
SB_BOTTOM	Derulează în dreapta jos.
SB_ENDSCROLL	Încheie derularea.
SB_LINELEFT	Derulează în stânga cu o unitate.
SB_LINERIGHT	Derulează în dreapta cu o unitate.
SB_PAGELEFT	Derulează în stânga cu lățimea ferestrei.
SB_PAGERIGHT	Derulează în dreapta cu lățimea ferestrei.
SB_THUMBPOSITION	Derulează la poziția absolută (un deplasament numeric exact, începând de la marginea din stânga a ferestrei). Poziția curentă este dată de parametrul *nPos* (cuvântul mai semnificativ din *wParam*).
SB_THUMBTRACK	Deplasează caseta de derulare la poziția specificată. Poziția curentă este precizată de parametrul *nPos* (cuvântul mai semnificativ din *wParam*).
SB_TOP	Derulează în stânga sus.

Tabelul 1355.2 *Valorile posibile pentru cuvântul mai puțin semnificativ al parametrului **wParam**.*

La fel ca și mesajul *WM_VSCROLL*, cuvântul mai semnificativ din *wParam* specifică poziția curentă a casetei de derulare dacă parametrul *nScrollCode* ia valorile *SB_THUMBPOSITION* sau *SB_THUMBTRACK;* altfel, cuvântul mai semnificativ conține o valoare inutilizabilă. În sfârșit, parametrul *lParam* returnează identificatorul controlului bară de derulare, dacă bara de derulare transmite mesajul. Dacă bara de derulare nu trimite mesajul, *lParam* este NULL.

Observați că atât mesajele *WM_VSCROLL*, cât și *WM_HSCROLL* conțin date cu poziția casetei de derulare numai pe 16 biți. În consecință, aplicațiile care se bazează numai pe *WM_HSCROLL* și *WM_VSCROLL* pentru datele cu poziția de derulare vor avea o valoare maximă de 65.535. Deoarece însă funcțiile *SetScrollPos, SetScrollRange, GetScrollPos* și *GetScrollRange* acceptă date cu poziția barei de derulare pe 32 de biți, există o stratagemă de a ocoli bariera de 16 biți a mesajelor *WM_HSCROLL* și *WM_VSCROLL*. Vezi fișierul help al compilatorului pentru informații suplimentare în legătură cu strategiile de ocolire a barierei de 16 biți.

1356 ACTIVAREA ȘI DEZACTIVAREA BARELOR DE DERULARE



Așa cum ați învățat, programele dumneavoastră pot controla corect și în mod complex barele de derulare. Una dintre cele mai obișnuite acțiuni executate de programele dumneavoastră cu barele de derulare este activarea și dezactivarea lor. Așa cum ați văzut în secțiunea 1352, programul poate dezactiva o bară de derulare dacă utilizatorul redimensionează fereastra la o dimensiune suficient de mare pentru a afișa complet tot ce conține fereastra. Pentru a activa și dezactiva bare de derulare, puteți folosi funcția *EnableScrollBar* care activează sau dezactivează una sau ambele săgeți ale barei. Prototipul funcției *EnableScrollBar* este următorul:

```
BOOL EnableScrollBar(
    HWND hWnd,   // identificator pentru fereastra sau bara
                 // de derulare
    UINT wSBflags, // indicator cu tipul barei de derulare
```

```
    UINT wArrows // indicator pentru sageata barei de derulare
);
```

Aşa cum vedeţi, funcţia *EnableScrollBar* acceptă trei parametri. Parametrul *hWnd* identifică controlul fereastră sau bară de derulare, în funcţie de parametrul *wSBflags*. Parametrul *wSBflags* specifică tipul barei de derulare. Acest parametru poate lua valorile prezentate în Tabelul 1356.1

Valoare	Semnificaţie
SB_BOTH	Activează sau dezactivează săgeţile în barele orizontală şi verticală asociate ferestrei respective. Parametrul *hWnd* trebuie să fie identificatorul ferestrei.
SB_CTL	Identifică bara de derulare ca un control bară de derulare. Parametrul *hWnd* trebuie să fie identificatorul controlului bară de derulare.
SB_HORZ	Activează sau dezactivează săgeţile pe bara de derulare orizontală asociată cu fereastra specificată. Parametrul *hWnd* trebuie să fie identificatorul ferestrei.
SB_VERT	Activează sau dezactivează săgeţile pe bara de derulare verticală asociată cu fereastra specificată. Parametrul *hWnd* trebuie să fie identificatorul ferestrei.

Tabelul 1356.1 *Valorile acceptate ale parametrului* ***wSBflag***.

În sfârşit, parametrul *wArrows* specifică dacă săgeţile barei de derulare sunt activate sau dezactivate şi precizează ce săgeată trebuie să activeze sau să dezactiveze funcţia *EnableScrollBar*. Parametrul *wArrows* va lua una din valorile prezentate în Tabelul 1356.2.

Valoare	Semnificaţie
ESB_DISABLE_BOTH	Activează ambele săgeţi de pe bara de derulare.
ESB_DISABLE_DOWN	Dezactivează săgeata de jos a unei bare de derulare verticale.
ESB_DISABLE_LEFT	Dezactivează săgeata din stânga a unei bare de derulare orizontale.
ESB_DISABLE_LTUP	Dezactivează săgeata din stânga a unei bare de derulare orizontale sau săgeata de sus a unei bare de derulare verticale.
ESB_DISABLE_RIGHT	Dezactivează săgeata din dreapta a unei bare de derulare orizontale.
ESB_DISABLE_RTDN	Dezactivează săgeata din dreapta a unei bare de derulare orizontale sau săgeata de jos a unei bare de derulare verticale.
ESB_DISABLE_UP	Dezactivează săgeata de sus a unei bare de derulare verticale.
ESB_ENABLE_BOTH	Activează ambele săgeţi ale unei bare de derulare.

Tabelul 1356.2 *Valorile acceptate de parametrul* ***wArrows***.

Dacă funcţia reuşeşte să activeze sau să dezactiveze săgeţile specificate în parametrul *wArrows*, valoarea returnată este diferită de zero. Dacă săgeţile sunt deja în starea solicitată sau intervine vreo eroare, valoarea returnată este zero.

CD-ROM-ul care însoţeşte cartea de faţă cuprinde programul *Enable_Disable.cpp* care activează sau dezactivează o bară de derulare verticală în funcţie de dimensiunea şi conţinutul ferestrei. Veţi observa modul în care programul *Enable_Disable* testează în funcţia *WndProc* dimensiunea şi conţinutul ferestrei, de fiecare dată când fereastra primeşte o

comandă *WM_SIZE* și activează sau, respectiv, dezactivează corespunzător barele de derulare, după ce fereastra a primit și programul a prelucrat mesajul *WM_SIZE*.

1357 UTILIZAREA FUNCȚIEI SCROLLDC

C/C++

În toate secțiunile anterioare ați învățat despre mai multe funcții ale barelor de derulare și despre mesajele generate de ele. Dar, pentru că nu ați învățat încă despre contextele de dispozitiv, este important să aflați cum pot programele dumneavoastră să deruleze ferestre care conțin grafică sau alte obiecte ne-textuale. Puteți, de asemenea, utiliza contextele de dispozitiv când desenați un text în fereastră, deși în general veți utiliza indirectarea pentru a accesa contextul. Când efectuați o derulare a contextului de dispozitiv într-o fereastră, veți prelucra în mod diferit derularea contextului de dispozitiv sau a unei părți a contextului de dispozitiv. Funcția *ScrollDC* derulează pe orizontală și verticală un dreptunghi de biți într-un context de dispozitiv. Veți utiliza funcția *ScrollDC* cu prototipul descris mai jos:

```
BOOL ScrollDC(
   HDC hDC,                    // identificator pentru contextul
                               // de dispozitiv
   int dx,                     // unitati de derulare orizontala
   int dy,                     // unitati de derulare verticala
   CONST RECT *lprcScroll,     // adresa structurii pentru
                               // dreptunghiul de derulare
   CONST RECT *lprcClip,       // adresa structurii pentru
                               // dreptunghiul de decupare
   HRGN hrgnUpdate,            // identificator al regiunii de
                               // derulare
   LPRECT lprcUpdate           // adresa structurii cu
                               // dreptunghiul actualizat
);
```

Funcția *ScrollDC* acceptă parametrii descriși în Tabelul 1357.

Parametru	Descriere
hDC	Identifică un context de dispozitiv care conține biții de derulat.
dx	Specifică volumul, în unități de dispozitiv, al derulării orizontale. Acest parametru trebuie să posede o valoare negativă pentru a derula la stânga.
dy	Specifică volumul, în unități de dispozitiv, al derulării verticale. Acest parametru trebuie să posede o valoare negativă pentru a derula în sus.
lprcScroll	Indică o structură care conține coordonatele dreptunghiului de derulat.
lprcClip	Indică o structură care conține coordonatele dreptunghiului de decupare. Sunt afectați numai biții de dispozitiv din dreptunghiul de decupare. Windows va desena biții derulați din afara dreptunghiului înspre interiorul lui. Windows nu va desena biții derulați dinspre interiorul dreptunghiului spre exteriorul lui.
hrgnUpdate	Identifică regiunea neacoperită de procesul de derulare. *ScrollDC* definește această regiune; nu este în mod necesar un dreptunghi.

Parametru	Descriere
lprcUpdate	Indică o structură *RECT* care primește coordonatele dreptunghiului care limitează regiunea de derulare actualizată. Aceasta este cea mai mare suprafață dreptunghiulară care necesită redesenarea. Când funcția returnează controlul, valorile din structură sunt exprimate în coordonate de client, indiferent de modul de mapare pentru contextul de dispozitiv specificat. Acesta permite aplicațiilor să utilizeze regiunea actualizată într-un apel la funcția *InvalidateRgn*, dacă este necesar.

***Tabelul 1357** Parametrii funcției **ScrollDC**.*

Dacă parametrul *lprcUpdate* este NULL, Windows nu calculează dreptunghiul actualizat. Dacă parametrii *hrgnUpdate* și *lprcUpdate* sunt ambii NULL, Windows nu calculează regiunea actualizată. Dacă *hrgnUpdate* nu este NULL, Windows va proceda ca și cum ar conține un identificator valid al regiunii neacoperite de procesul de derulare, definit de *ScrollDC*. Pentru a înțelege mai bine procesul executat de funcția *ScrollDC*, să analizăm programul *Scroll_DC.cpp* din CD-ROM-ul care însoțește cartea de față. Programul *Scroll_DC* desenează o serie de linii în fereastră. Când încercați să derulați fereastra cu bara de derulare, programul *Scroll_DC* va derula numai o secțiune a ferestrei.

Chiar dacă programul *Enable_Disable.cpp* nu este probabil suficient de clar, deoarece nu ați învățat încă despre contextele de dispozitiv, prelucrarea finală pe care o face funcția (de derulare a porțiunii ferestrei) este simplă. Programul *ScrollDC* generează un context de dispozitiv care va stabili regiunea pe care *ScrollDC* o modifică cu 20 de unități mai puțin decât zona client a ferestrei. Apoi programul apelează funcția *ScrollDC* pentru a derula fereastra spre dreapta.

MODELUL DE MEMORIE WIN32 — C/C++ 1358

Așa cum ați aflat din secțiunile anterioare, modelul de memorie Win32 simplifică mult gestionarea memoriei. Modelele de memorie *small*, *large*, *huge* nu mai există în sistemul Windows pe 32 de biți. Deoarece nu mai există modele de memorie, programele Win32 nu mai fac distincție între memoria *near* și *far*. Fără segmente, programul și datele rezidă acum în aceeași memorie liniară, care manevrează mai ușor programele și blocurile mari de date.

În mediul Win32, fiecare proces are propriul spațiu de adresă virtual pe 32 de biți, de până la patru gigaocteți (4GB). Windows pune la dispoziția utilizatorului primii 2 gigaocteți în memoria inferioară (de la 0x00000000 la 0x7FFFFFFF) și rezervă doi gigaocteți pentru nucleul sistemului de operare în memoria superioară (de la 0x80000000 la 0xFFFFFFFF). Adresele utilizate de procese nu mai reprezintă locații fizice reale în memorie. În schimb nucleul sistemului de operare (software cheie al sistemului de operare care controlează CPU, firele de execuție, memoria etc.) întreține o pagină de mapare pentru fiecare proces, pe care Windows o utilizează pentru a converti adresele virtuale în adrese fizice corespunzătoare. Procesele nu pot scrie în afara propriului spațiu de prelucrare (ceea ce protejează procesele unele față de altele).

Interfața Win32 API acceptă funcțiile de alocare *GlobalAlloc* și *LocalAlloc*. În mediul Win32 alocările globale și locale sunt de fapt același lucru. În mediul Win16 alocarea locală se face în cadrul spațiului de adresă al procesului, iar alocarea globală se face în afara spațiului adresă al procesului. În mediul Win32, Windows alocă ambele tipuri de memorie în cadrul spațiului de adresă al procesului și ambele tipuri de memorie sunt accesibile din programele

dumneavoastră utilizând pointeri pe 32 de biți. Așa cum veți învăța, însă, utilizarea alocării locale poate face programele dumneavoastră mai ușor de înțeles.

Mediul Win32 introduce două noi modalități de gestionare a memorie de către programele dumneavoastră: gestionarul de memorie virtuală și gestionarul de memorie heap locală. Funcțiile API de gestionare a memoriei virtuale sunt similare funcțiilor API de gestionare a memoriei globale, cu excepția faptului că programele dumneavoastră pot rezerva blocuri mari de memorie virtuală și să aloce mai târziu aceste blocuri de memorie virtuală. Noul tip de gestionar de memorie heap diferă de gestionarul de memorie heap utilizat până acum, în sensul că noul gestionar permite programelor să creeze memorii heap multiple și separate. Memoriile heap multiple vă pun la dispoziție modalități simple și eficiente de alocare a unor cantități mici de memorie (cum ar fi memoria necesară unui singur pointer). În următoarele câteva secțiuni veți învăța mai mult despre gestionara memoriei Windows. Veți învăța, de asemenea, despre unele funcții pe care le puteți utiliza pentru gestionarea mai eficientă a memoriei Windows globale și virtuale.

1359 MEMORIA GLOBALĂ ȘI LOCALĂ



Așa cum ați învățat în secțiunea 1358, în modelul de memorie liniară pe 32 de biți al aplicațiilor Win32, Windows (deci și programele dumneavoastră) nu face distincție între memoria locală și memoria globală. În consecință, Windows (și deci programele dumneavoastră) nu fac distincție între memoria heap globală și cea locală. Deci obiectele din memorie alocate de program cu *GlobalAlloc* și *LocalAlloc* sunt al fel. Windows alocă în pagini private și angajate de memorie (adică pagini de memorie accesibile altor programe) cu drept la scriere/citire. *Memoria privată* este memoria care nu poate fi accesată de alte programe, nici de programele concurente în execuție.

Funcțiile *GlobalAlloc* și *LocalAlloc* pot aloca un bloc de memorie de orice dimensiune reprezentat pe 32 de biți care concordă cu memoria disponibilă (inclusiv volumul de stocare disponibil în fișierul de pagini, despre care veți învăța în continuare). Obiectele de memorie alocate de program pot fixe (în anumite locații de memorie) sau mobile. Dacă alocați un obiect fix de memorie, el va rezida într-o locație dată de memorie fizică pe tipul vieții sale.

Puteți marca obiectele mobile de memorie ca obiecte descărcabile. În Windows 3.x obiectele de memorie mobile erau importante pentru gestionarea memoriei. Pe de altă parte, în Win32 programele dumneavoastră vor utiliza memorie virtuală. Sistemul va putea deci gestiona memoria fără nici un impact asupra adreselor de memorie virtuală. Când programul dumneavoastră mută o pagină de memorie, Windows mapează simplu pagina virtuală a procesului la o nouă locație. Programele dumneavoastră vor utiliza memorie mobilă pentru alocarea memorie descărcabile, pe care programele o vor utiliza și descărca regulat (matrice mici, pointeri etc).

1360 MEMORIA VIRTUALĂ



În secțiunile precedente ați aflat despre conceptul de memorie virtuală. Așa cum precizează secțiunea 1359, gestionarul de memorie virtuală permite programelor dumneavoastră să trateze volumul limitat de memorie fizică a calculatorului și dimensiunea (de regulă, mare) a fișierului de paginare ca un singur bloc de memorie contiguă. Pentru a permite programului să gestioneze memoria în acest mod, gestionarul de memorie virtuală lucrează cu *mapări* ale memoriei reale și nu cu memoria reală propriu-zisă. Pentru că lucrați numai cu iluzia unei

memorii contigue și nu cu blocuri de memorie contiguă, cei care au proiectat sistemul Windows au numit acest model de memorie *model de memorie virtuală*. Figura 1360 prezintă modelul logic după care Windows folosește memoria virtuală pentru a accesa memoria fizică.

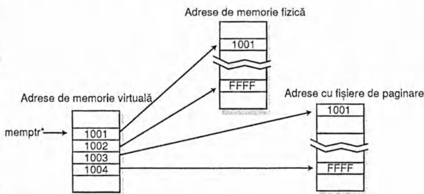


Figura 1360 *Utilizarea memoriei virtuale pentru a accesa memoria fizică.*

Datorită modului în care Windows gestionează memoria virtuală, spațiul de adresă virtuală pentru fiecare proces este mult mai mare decât adresa fizică totală disponibilă tuturor proceselor. Pentru a crește spațiul de memorare, Windows utilizează hard-discul pentru stocări suplimentare. Volumul total de memorie disponibilă pentru toate procesele este suma dintre memoria fizică spațiul liber disponibil pe disc în cadrul fișierului de paginare Windows. Fișierul de paginare este un fișier pe disc, utilizat de Windows pentru creșterea memoriei efective a calculatorului. Windows organizează spațiul de memorie virtuală în pagini sau unități de memorie. Dimensiunea paginii depinde de calculatorul gazdă. Puteți invoca funcția *GetSystemInfo* pentru a determina dimensiunea paginii unui calculator. Pe calculatoarele x86, dimensiunea paginii este de 4Kb.

În modelul de memorie Win32, sistemul de operare pune la dispoziție un spațiu de adresă privată pentru fiecare proces. Când un fir de execuție al unui proces rulează, acel fir poate avea acces numai la memoria care aparține procesului. Memoria care aparține tuturor celorlalte procese este ascunsă și inaccesibilă firului de execuție. Deoarece fiecare proces are propriul spațiu de adresă virtuală de 4Gb, fiecare proces interpretează memoria ca și cum ea ar rula de la adresa 0x00000000 la adresa 0xFFFFFFFF. Astfel, două procese care rulează simultan ar putea fiecare să stocheze memorie la adresa 0x12341234 fără să interfereze unul cu celălalt. În mod clar, această partajare de adresă nu poate fi posibilă dacă ambele procese utilizează același spațiu de adresă.

În realitate, gestionarul de memorie virtuală mapează adresa virtuală la o adresă fizică reală – care poate fi în memoria fizică a calculatorului sau în fișierul de paginare. Important este că, în ceea ce privește aplicațiile, adresa memoriei este 0x12341234, dacă memoria reală este la locația RAM 0x00012345, sau în sectorul 14352 al hard-discului. În consecință, gestionarul de memorie virtuală vă permite executarea simultană a mai multor programe, fără să vă preocupe dacă fiecare program nu va supraîncărca memoria altui program în execuție.

În Windows 95, sistemul de operare împarte spaţiul de adrese virtuale de 4Gb pentru fiecare proces în patru partiţii. Windows 95 utilizează partiţia de la 0x00000000 la 0x003FFFFF (o partiţie de 4Mb la limita de jos a spaţiului de adresă virtuală) pentru a menţine compatibilitatea cu MS-DOS şi Windows pe 16 biţi. Aplicaţiile Win32 nu trebuie să citească sau să scrie în această partiţie. Dacă aplicaţia dumneavoastră încearcă să acceseze această memorie, sistemul de operare va returna un pointer NULL şi va deveni instabil (cu opriri de program, blocarea sistemului de operare etc.)

Windows 95 utilizează partiţia de la 0x00400000 la 0x7FFFFFFF pentru spaţiu de adresă privat şi nepartajat. Procesele Win32 nu pot citi, scrie sau accesa în orice fel datele altui proces stocate în această partiţie de aproape 2Gb. Pentru aplicaţiile dumneavoastră trebuie să menţineţi grosul informaţiilor de proces în cadrul acestei partiţii.

Windows 95 utilizează ceilalţi 2Gb de spaţiu de adresă virtuală a fiecărui proces pentru a stoca fişierele partajate şi fişierele sistemului de operare. Când operaţi cu memoria din spaţiul de adresă al programului dumneavoastră, va trebui să evitaţi accesarea spaţiului de adresă mai sus de 0x80000000, cu excepţia cazului în care programul dumneavoastră nu accesează intenţionat un fişier partajat sau un fişier de sistem.

1361 DIN NOU DESPRE MEMORIA HEAP



Aşa cum aţi învăţat în secţiunea 1360, Windows alocă programelor dumneavoastră 4Gb de spaţiu de adresă virtuală pentru fiecare proces care rulează. Funcţiile Win32 de heap permit proceselor să creeze un *heap privat*, care este un bloc de una sau mai multe pagini în spaţiul de adresă al procesului. Funcţia *HeapCreate* produce un heap de o dimensiune dată, iar funcţiile *HeapAlloc* şi *HeapFree* alocă şi eliberează memorie din heap. Când creaţi memorii heap în programele Windows, sistemul va crea memoria heap începând de la 0x7FFFFFFF în jos, în limita de 2Gb din memoria rezervată procesului.

Obiectele din heap pot creşte dinamic în intervalul pe care îl specificaţi la creare cu funcţia *HeapCreate*. Dimensiunea maximă a memoriei heap determină numărul de pagini de memorie pe care ea o rezervă. Dimensiunea iniţială determină numărul de pagini angajate, pagini cu drept de citire/scriere pe care Windows le alocă iniţial în heap. Windows alocă automat pagini suplimentare din spaţiul rezervat dacă solicitările funcţiei *HeapAlloc* trec dincolo de dimensiunea curentă a paginilor angajate. După ce Windows angajează pagini la un heap, el nu va elibera aceste pagini angajate decât dacă procesul se încheie sau programul distruge memoria heap cu funcţia *HeapDestroy*. Deoarece memoria alocată în heap cu *HeapAlloc* are o locaţie fixă în spaţiul de adresă virtuală a procesului, iar sistemul nu poate compacta memoria heap, trebuie să scrieţi în aşa fel aplicaţiile, încât să fragmenteze cât mai puţin memoria heap.

Memoria heap privată este accesibilă numai procesului care a produs acel heap. Dacă o bibliotecă cu legare dinamică (DLL) produce un heap privat, Windows produce memoria heap privată în spaţiul de adresă al procesului care a apelat biblioteca de legare dinamică (în Windows 95, peste 0x80000000). Însă numai procesul care a apelat biblioteca de legare dinamică poate accesa informaţia din memoria heap privată a bibliotecii respective, aceasta însemnând că mai multe procese care execută aceeaşi bibliotecă de legare dinamică pot crea mai multe memorii heap private asociate acestor biblioteci, dar fiecare instanţă a bibliotecii poate accesa un singur heap privat.

ALOCAREA UNUI BLOC DE MEMORIE ÎN MEMORIA HEAP GLOBALĂ

C/C++1362

Programele dumneavoastră pot utiliza mai multe tehnici diferite de alocare a memoriei în mediile Windows 95 sau Windows NT. Una dintre cele mai utilizate modalităţi este alocarea memoriei din memoria heap globală, asemănătoare alocării cu funcţiile *malloc* şi *halloc* executate în programele DOS. Veţi utiliza funcţia *GlobalAlloc* pentru a aloca memorie din memoria heap globală. Funcţia *GlobalAlloc* alocă numărul de octeţi specificaţi din heap. Prototipul funcţiei *GlobalAlloc* este prezentat mai jos:

```
HGLOBAL GlobalAlloc(
    UINT uFlags, // atributele alocarii
    DWORD dwBytes // numar de octeti alocati
);
```

Parametrul *uFlags* specifică modul de alocare a memoriei. Dacă este zero, valoarea implicită este *GMEM_FIXED*. Cu excepţia combinaţiilor incompatibile care sunt precis indicate, poate fi folosită orice combinaţie de indicatoare prezentate în Tabelul 1362.1. Pentru a indica dacă funcţia alocă memorie fixă sau deplasabilă, folosiţi una din valorile prezentate în tabelul menţionat.

Indicator	Semnificaţie
GMEM_FIXED	Alocă memorie fixă. Acest indicator nu poate fi combinat cu *GMEM_MOVEABLE* sau *GMEM_DISCARDABLE*. Valoarea returnată este un pointer la blocul de memorie. Pentru a accesa memoria, procesul apelant foloseşte valoarea returnată ca un pointer.
GMEM_MOVEABLE	Alocă memorie deplasabilă. Acest indicator nu poate fi combinat cu *GMEM_FIXED*. Valoarea returnată este identificatorul obiectului de memorie. Identificatorul este o cantitate reprezentată pe 32 de biţi, privată pentru procesul apelant. Pentru a converti identificatorul într-un pointer, se utilizează funcţia *GlobalLock*.
GPTR	Combină indicatoarele *GMEM_FIXED* şi *GMEM_ZEROINIT* din Tabelul 1362.2.
GHND	Combină indicatoarele *GMEM_MOVEABLE* şi *GMEM_ZEROINIT* din Tabelul 1362.2.

***Tabelul 1362.1** Tipuri de alocare a memoriei utilizate cu funcţia* **GlobalAlloc**.

În plus faţă de valorile specificate în Tabelul 1362.1, parametrul *uFlags* poate combina oricare din valorile prezentate în Tabelul 1362.2, cu excepţia cazurilor de combinaţii incompatibile indicate anume în tabel.

Valoare	Semnificație
GMEM_DDESHARE	Alocă memorie pentru funcțiile de schimb de date dinamice (DDE) din conversațiile DDE. Acest indicator este disponibil în scopul compatibilității cu Win16. Poate fi utilizat de unele aplicații pentru a optimiza performanțele operațiilor DDE și trebuie deci specificat dacă memoria urmează a fi utilizată de DDE. Numai procesele care utilizează DDE sau memoria clipboard pentru comunicații între procese trebuie să specifice acest indicator.
GMEM_DISCARDABLE	Alocă memorie descărcabilă (memorie care nu este fixată la o anumită adresă în spațiul de adresă virtuală al procesului). Acest indicator nu poate fi combinat cu *GMEM_FIXED*. Unele aplicații Win32 ignoră acest indicator.
GMEM_LOWER	Ignorat de Win32. Acest indicator este oferit numai pentru compatibilitate cu versiunile Windows 3.x.
GMEM_NOCOMPACT	Nu compactează sau descarcă memoria pentru a satisface solicitarea de alocare.
GMEM_NODISCARD	Nu descarcă memoria pentru satisfacerea cererii de alocare
GMEM_NOT_BANKED	Ignorat de Win32. Acest indicator este pus la dispoziție numai pentru compatibilitate cu Windows 3.x.
GMEM_NOTIFY	Ignorat de Win32. Acest indicator este pus la dispoziție numai pentru compatibilitate cu Windows 3.x.
GMEM_SHARE	Alocă memorie ce va fi folosită de funcții DDE pentru o conversație DDE. Același cu *GMEM_DDESHARE*.
GMEM_ZEROINIT	Inițializează conținutul memoriei cu zero.

***Tabelul 1362.2** Valori suplimentare de indicatoare pentru funcția* **GlobalAlloc**.

Pe lângă specificarea tipului de memorie alocată de *GlobalAlloc*, programul trebuie să specifice parametrul *dwBytes* care conține numărul de octeți de alocat. Dacă parametrul este zero, iar parametrul *uFlags* specifică valoarea *GMEM_MOVEABLE*, funcția va returna un identificator al unui obiect de memorie marcat ca descărcabil. Dacă funcția reușește, valoarea returnată este identificatorul noului obiect de memorie alocat. Dacă funcția eșuează, valoarea returnată este NULL.

Dacă memoria heap nu conține suficient spațiu liber pentru a satisface cererea, *GlobalAlloc* returnează NULL. Deoarece NULL este utilizat pentru a semnala o eroare, nu este niciodată alocată adresa virtuală zero. Este, de aceea, ușor să detectezi utilizarea unui pointer NULL. Întreaga memorie creată de Windows are acces la execuție. Nu este necesară nici o funcție specială pentru executarea codului generat dinamic. Windows garantează că memoria alocată programului cu funcția *GlobalAlloc* se va încadra în limita de 8 octeți.

Windows limitează funcțiile *GlobalAlloc* și *LocalAlloc* la un număr total 65536 de identificatori combinați pe proces, pentru memoria alocată global cu *GMEM_MOVEABLE* și memoria alocată local cu *LMEM_MOVEABLE*. Această limitare nu se aplică memoriei *GMEM_FIXED* sau *LMEM_FIXED*. Dacă funcția *GlobalAlloc* reușește, ea alocă cel puțin cantitatea de memorie cerută. Când cantitatea reală alocată de *GlobalAlloc* este mai mare decât cantitatea cerută, procesul poate utiliza întreaga cantitate. Pentru a determina numărul real de octeți alocați de *GlobalAlloc*, se utilizează funcția *GlobalSize*.

Pentru a înțelege mai bine modul de operare al funcției *GlobalAlloc* parcurgeți programul *Global_Alloc.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *GlobalAlloc* alocă memorie pentru a stoca un șir. Când aplicația începe, tratarea mesajului *WM_CREATE* produce un buffer de 27 de caractere (26 de octeți și un terminator NULL). Când utilizatorul a selectat *Test!* programul afișează bufferul și dimensiunea sa.

UTILIZAREA FUNCȚIEI GLOBALREALLOC PENTRU A SCHIMBA DINAMIC DIMENSIUNEA MEMORIEI HEAP

C/C++ 1363

Așa cum ați învățat în secțiunea 1362, programul dumneavoastră poate utiliza funcția *GlobalAlloc* pentru a aloca memorie din memoria heap globală. Deseori însă, programul trebuie să realoce un bloc de memorie după alocarea sa inițială. Puteți proceda astfel cu funcția *GlobalReAlloc* care schimbă dimensiunea sau atributele unui obiect de memorie globală specificat. În funcție de modul de invocare, dimensiunea poate crește sau diminua. Prototipul funcției *GlobalReAlloc* este prezentat mai jos:

```
HGLOBAL GlobalReAlloc(
    HGLOBAL hMem,      // identificator pentru obiectul de
                       // memorie globala
    DWORD dwBytes,     // noua dimensiune a blocului
    UINT uFlags        // modul de realocare a obiectului
);
```

Identificatorul *hMem* identifică obiectul de memorie globală care se realocă. Acest identificator este returnat fie de funcția *GlobalAlloc*, fie de *GlobalReAlloc*. Parametrul *dwBytes* specifică dimensiunea nouă, în octeți, a blocului de memorie. Dacă parametrul este zero și parametrul *uFlags* specifică valoarea *GMEM_MOVEABLE*, funcția returnează identificatorul blocului de memorie marcat ca descărcabil. Dacă *dwBytes* este zero și *uFlags* specifică valoarea *GMEM_MODIFY*, funcția API ignoră parametrul *dwBytes*. Parametrul *uFlags* specifică modul de realocare a obiectului de memorie globală. Dacă *uFlags* specifică valoarea *GMEM_MODIFY*, parametrul modifică atributele obiectului de memorie, iar parametrul *dwBytes* este ignorat. În caz contrar, acest parametru controlează realocarea obiectului de memorie.

Când apelați funcția *GlobalReAlloc*, programul poate combina valoarea *GMEM_MODIFY* cu una sau ambele valori din Tabelul 1363.1.

Fanion	Semnificație
GMEM_DISCARDABLE	Alocă memorie descărcabilă dacă este specificat, de asemenea, indicatorul *GMEM_MODIFY*. Acest indicator este ignorat dacă obiectul nu a fost anterior alocat ca deplasabil sau dacă a fost, de asemenea, specificat indicatorul *GMEM_MOVEABLE*.
GMEM_MOVEABLE	Numai pentru Windows NT. Schimbă un obiect fix de memorie într-unul mobil, dacă este de asemenea specificat indicatorul *GMEM_MODIFY*.

Tabelul 1363.1 *Valorile compatibile cu indicatorul* ***GMEM_MODIFY***.

Dacă *uFlags* nu specifică *GMEM_MODIFY*, el poate fi orice combinație a următoarelor indicatoare:

Indicator	Semnificație
GMEM_MOVEABLE	Dacă *dwByte* este zero, descarcă un bloc de memorie anterior deplasabil și descărcabil. Dacă numărul de blocare al obiectului nu este zero sau dacă blocul nu este deplasabil sau descărcabil, funcția eșuează. Dacă *dwBytes* are o valoare diferită de zero, permite sistemului să deplaseze blocul realocat la o nouă locație, fără să schimbe atributele de deplasabil sau fixat ale obiectului. Dacă obiectul este fixat, identificatorul returnat poate fi diferit de identificatorul specificat de parametrul *hMem*. Dacă obiectul este deplasabil, blocul poate fi deplasat fără invalidarea identificatorului, chiar dacă obiectul este la acel moment blocat de un apel anterior la funcția *GlobalLock*. Pentru obținerea noii adrese a blocului de memorie, utilizați funcția *GlobalLock*.
GMEM_NOCOMPACT	Previne compactarea sau descărcarea memoriei pentru satisfacerea cererii de alocare.
GMEM_ZEROINIT	Conținutul suplimentar de memorie va putea să fie inițializat cu zero de Windows, dacă obiectul de memorie crește în dimensiune.

Tabelul 1363.2 *Valori suplimentare ale parametrului* ***uFlags***.

Dacă funcția reușește, valoarea returnată va fi identificatorul obiectului de memorie realocat. Dacă funcția eșuează, valoarea returnată va fi NULL. Dacă funcția *GlobalReAlloc* realocă un obiect deplasabil, valoarea returnată este identificatorul obiectului de memorie. Pentru convertirea unui identificator de pointer, folosiți funcția *GlobalLock*. Dacă *GlobalReAlloc* realocă un obiect fix, valoarea identificatorului returnat este adresa primului octet de blocului de memorie. Pentru a accesa memoria, procesele pot simplu să folosească valoarea returnată ca un pointer. Dacă funcția *GlobalReAlloc* eșuează, memoria inițială nu este eliberată, iar identificatorul și pointerul inițiali vor fi încă valizi.

Pentru a înțelege mai bine modul de operare al funcției *GlobalReAlloc*, parcurgeți programul *Global_ReAlloc.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Global_ReAlloc* execută procese asemănătoare cu programul *GlobalAlloc*. Însă, programul *Global_ReAlloc* realocă alți 27 de octeți de memorie pentru a prezenta alfabetul în litere mici, atunci când se selectează opțiunea *Test*.

1364 DESCĂRCAREA UNUI BLOC DE MEMORIE ALOCATĂ



În secțiunile anterioare ați utilizat funcțiile membre *free* și *delete*, pentru a descărca memoria alocată în program. Când alocați sau realocați memorie din memoria heap globală, programul dumneavoastră trebuie să utilizeze funcția *GlobalDiscard* pentru descărcarea memoriei după terminarea prelucrărilor în memorie. Funcția *GlobalDiscard* descarcă blocul de memorie globală alocat anterior cu indicatorul *GMEM_DISCARDABLE*. Numărul de blocări ale obiectului de memorie pe care vreți să-l descărcați trebuie să fie zero, altfel funcția nu va putea descărca memorie. Prototipul funcției *GlobalDiscard* este prezentat mai jos:

```
HGLOBAL GlobalDiscard(
   HGLOBAL hglbMem    // identificator pentru obiectul de
                      // memorie globala
);
```

Parametrul *hglbMem* identifică obiectul de memorie globală de descărcat. Dacă funcția reușește, valoarea returnată va fi identificatorul obiectului de memorie (adică, *hglbMem*). Dacă funcția eșuează, valoarea returnată va fi NULL.

Funcția *GlobalDiscard* descarcă numai obiecte globale alocate de procesul apelant cu valoarea *GMEM_DISCARDABLE*. Dacă procesul încearcă să descarce un obiect fix sau blocat, funcția eșuează. Deși funcția *GlobalDiscard* descarcă blocul de memorie al obiectului, identificatorul obiectului rămâne valid. Procesul poate după aceea să transmită identificatorul funcției *GlobalReAlloc* pentru a aloca un alt bloc de memorie identificat de același identificator.

UTILIZAREA FUNCȚIEI GLOBALFREE

C/C++1365

În secțiunea 1364, ați învățat despre funcția *GlobalDiscard* pe care programele dumneavoastră o va utiliza pentru descărcarea unui bloc de memorie anterior alocat și a menține un identificator pentru blocul de memorie pentru o posibilă utilizare viitoare. Pe de altă parte, dacă știți că programul nu va reutiliza același bloc de memorie, dacă vreți să evitați ca programul să reutilizeze același bloc de memorie sau dacă nu sunteți sigur că programul dumneavoastră a alocat blocul cu valoarea *GMEM_DISCARDABLE*, programul poate utiliza funcția *GlobalFree* pentru a elibera un obiect de memorie. Funcția *GlobalFree* eliberează obiectul de memorie globală specificat și îi invalidează identificatorul. Prototipul funcției *GlobalFree* este prezentat mai jos:

```
HGLOBAL GlobalFree(HGLOBAL hMem);
```

Parametrul *hMem* identifică obiectul de memorie globală. Spre deosebire de funcția *GlobalDiscard*, dacă funcția *GlobalFree* reușește, valoarea returnată este NULL. Dacă funcția eșuează, valoarea returnată este egală cu identificatorul obiectului de memorie globală.

Este posibil să apară coruperea memoriei heap sau o eroare de violare a accesului (*EXCEPTION_ACCESS_VIOLATION*), în cazul în care procesul încearcă să examineze sau să modifice memoria după ce a fost eliberată. Dacă parametrul *hgblMem* este NULL, *GlobalFree* eșuează și sistemul generează o eroare de violare a accesului. Atât *GlobalFree*, cât și *LocalFree* vor elibera un obiect de memorie blocat. Obiectul de memorie blocat are numărul de blocare mai mare decât zero. Funcția *GlobalFree* blochează obiectul de memorie globală (care împiedică altă funcție să descarce obiectul de memorie) și incrementează numărul de blocare cu unu. Funcția *GlobalUnlock* îl deblochează și decrementează numărul de blocare cu unu. Pentru a obține numărul de blocare al obiectului de memorie globală, utilizați funcția *GlobalFlags*.

Observație: *Sub Windows NT însă, în cazul în care aplicația rulează sub o versiune de depanare (DBG) de Windows NT, cum e cea distribuită cu CD-ROM-ul SDK, atât funcția* ***GlobalFree****, cât și* ***LocalFree*** *plasează un punct de întrerupere exact înainte de eliberarea unui obiect blocat. Aceasta permite programatorului să testeze de două ori comportamentul intenționat. Apăsarea tastei G în timpul utilizării depanatorului în această situație permite executarea operațiunii de eliberare.*

1366 UTILIZAREA FUNCȚIILOR GLOBALLOCK ȘI GLOBALHANDLE

C/C++

Așa cum ați învățat, programele dumneavoastră pot utiliza funcțiile *GlobalAlloc* și *GlobalReAlloc* pentru alocarea memoriei din zona heap globală. Așa cum ați constatat însă, ambele funcții de alocare returnează un identificator al memoriei alocate. Pe de altă parte, majoritatea programelor dumneavoastră pot să utilizeze memoria cu un pointer. Puteți folosi funcțiile *GlobalLock* și *GlobalHandle* pentru a converti ușor memoria alocată la un pointer și înapoi la un dentificator. Funcția *GlobalLock* blochează un obiect de memorie globală și returnează un pointer la primul octet al blocului de memorie al obiectului. Blocul de memorie asociat cu un obiect de memorie blocat nu poate fi deplasat sau descărcat. Pentru blocurile de memorie alocate cu *GMEM_MOVEABLE*, funcția incrementează numărul de blocare asociat cu obiectul de memorie. Prototipul funcției *GlobalLock* este prezentat mai jos:

```
LPVOID GlobalLock(
   HGLOBAL hMem // addresa obiectului de memorie globala
);
```

În funcția *GlobalLock*, parametrul *hMem* identifică obiectul de memorie globală. Acest identificator este returnat fie de funcția *GlobalAlloc*, fie de *GlobalReAlloc*. Dacă funcția *GlobalLock* reușește, valoarea returnată este un pointer la primul octet al blocului de memorie. Dacă funcția *GlobalLock* eșuează, valoarea returnată va fi zero.

Structurile interne de date pentru fiecare obiect de memorie cuprind un număr de blocare care este inițial zero. Pentru obiectele de memorie deplasabile, funcția *GlobalLock* incrementează numărul cu unu, iar funcția *GlobalUnlock* îl decrementează cu unu. Pentru fiecare apel pe care procesul îl face la *GlobalLock* pentru un obiect, el trebuie să apeleze în cele din urmă și *GlobalUnlock*. Memoria blocată nu poate fi deplasată sau descărcată, dacă obiectul de memorie nu este realocat utilizând funcția *GlobalReAlloc*. Blocul de memorie al unui obiect de memorie blocat, rămâne blocat până când numărul său de blocare este decrementat până la zero, moment în care el poate fi deplasat sau descărcat. Obiectele de memorie alocate anterior cu indicatorul *GMEM_FIXED* au întotdeauna numărul de blocare zero. Pentru aceste obiecte, valoarea pointerului returnat este egală cu valoarea identificatorului specificat. Dacă blocul de memorie respectiv a fost descărcat sau dacă blocul de memorie are do dimensiune de zero octeți, funcția returnează NULL. Obiectele descărcate au întotdeauna contorul de blocare zero. În general, veți implementa funcția *GlobalLock* într-un mod asemănător cu cel prezentat în următorul fragment:

```
HGLOBAL hMem = GlobalAlloc(GHND, 27);
LPSTR pMem;
if (hMem && (pMem = (LPSTR) GlobalLock(hMem)) !=NULL)
   ...
```

Codul de mai sus alocă un identificator la 27 de octeți de memorie (variabila *hMem*), apoi blochează memoria respectivă în pointerul *pMem*, care este simultan convertit la tipul șir de caractere.

Așa cum ați învățat, programele dumneavoastră vor utiliza deseori funcția *GlobalLock* pentru convertirea identificatorilor de memorie la pointeri. Veți utiliza de regulă funcția *GlobalHandle* pentru convertirea pointerului înapoi la un indentificator. Funcția *GlobalHandle* preia

identificatorul asociat cu pointerul specificat la blocul de memorie globală. Cel mai adesea veți folosi conversia unui pointer la un identificator pentru pregătirea comenzii *GlobalFree*. Veți utiliza funcția *GlobalHandle* cu prototipul prezentat mai jos:

```
HGLOBAL GlobalHandle(
  LPCVOID pMem // pointer la blocul de memorie globala
);
```

Parametrul *pMem* este un pointer la primul octet al blocului de memorie globală. Acest pointer este returnat de funcția *GlobalLock*. Dacă funcția *GlobalHandle* reușește, ea va returna identificatorul obiectului de memorie globală specificat. Dacă funcția *GlobalHandle* eșuează, valoarea returnată va fi NULL.

Când funcția *GlobalAlloc* alocă un obiect de memorie cu indicatorul *GMEM_MOVEABLE*, ea returnează identificatorul obiectului. Funcția *GlobalLock* convertește acest identificator într-un pointer la blocul de memorie, iar *GlobalHandle* convertește pointerul înapoi în identificator. O implementare generală a funcției *GlobalHandle* este prezentată în fragmentul de cod de mai jos:

```
HGLOBAL hMem = GlobalHandle(pMem);
GlobalUnlock(hMem);
pMem = NULL;
hMem = GlobalReAlloc(hMem, (26*2)+1, GMEN_MOVABLE);
```

În acest caz particular, programul creează identificatorul, apoi deblochează identificatorul (eliberând memoria). După aceea, programul realocă memorie, atât cât este necesar. Codul care urmează în programul dumneavoastră va reconverti probabil identificatorul în pointer.

TESTAREA MEMORIEI CALCULATORULUI

C/C++1367

Așa cum ați învățat, Windows menține informații atât despre memoria fizică a calculatorului, cât și despre memoria virtuală. Deseori, programele dumneavoastră vor necesita informații despre cantitatea disponibilă de memorie liberă care poate fi accesată de program. În acest scop se utilizează funcția *GlobalMemoryStatus* pentru regăsirea stării curente a memoriei calculatorului. Funcția *GlobalMemoryStatus* returnează informații despre memoria fizică și memoria virtuală. Prototipul funcției *GlobalMemoryStatus* este prezentat mai jos:

```
VOID GlobalMemoryStatus(
  LPMEMORYSTATUS lpBuffer        // pointer la structura starii
                                 // memoriei
);
```

Parametrul *lpBuffer* indică o structură *LPMEMORYSTATUS* în care este returnată informația despre disponibilitatea memoriei curente. Înaintea apelării acestei funcții, procesul apelant trebuie să stabilească membrul *dwLength* al structurii. Structura *MEMORYSTATUS* conține informația despre disponibilitatea memoriei curente. Windows API definește structura *MEMORYSTATUS* în modul prezentat mai jos:

```
typedef struct _MEMORYSTATUS {
  DWORD dwLength;           // dimensiunea lui MEMORYSTATUS
```

```
    DWORD dwMemoryLoad;        // procentul de memorie folosita
    DWORD dwTotalPhys;         // octeti de memorie fizica
    DWORD dwAvailPhys;         // octeti de memorie fizica libera
    DWORD dwTotalPageFile;     // octeti ai fisierului de paginare
    DWORD dwAvailPageFile;     // octeti liberi ai fisierului de
                               // paginare
    DWORD dwTotalVirtual;      // octeti ai spatiului de adresa
                               // pentru utilizator
    DWORD dwAvailVirtual;      // octeti liberi pentru utilizator
} MEMORYSTATUS;
```

Așa cum constatați, structura *MEMORYSTATUS* stochează informații importante despre memoria curentă disponibilă a calculatorului. Tabelul 1367 explică membrii structurii *MEMORYSTATUS*:

Membru	Semnificație
dwLength	Indică dimensiunea structurii. Procesul apelant trebuie să stabilească acest membru înaintea apelării funcției *GlobalMemoryStatus*.
dwMemoryLoad	Specifică un număr între 0 și 100 care dă o idee generală asupra utilizării memorie curente, unde 0 indică neutilizarea memoriei, iar 100 indică utilizarea maximă a memoriei.
dwTotalPhys	Indică numărul total de octeți de memorie fizică.
dwAvailPhys	Indică numărul de octeți de memorie fizică disponibilă.
dwTotalPageFile	Indică numărul total de octeți care pot fi stocați în fișierul de paginare. De reținut că acest număr nu reprezintă dimensiunea reală a unui fișier de paginare pe disc.
dwAvailPageFile	Indică numărul de octeți disponibili în fișierul de paginare.
dwTotalVirtual	Indică numărul total de octeți care pot fi descriși de Windows în partea modului utilizator din spațiul virtual de adresare a procesului apelant.
dwAvailVirtual	Indică numărul de octeți de memorie nerezervată și neangajată în partea modului utilizator din spațiul virtual de adresare a procesului apelant.

Tabelul 1367 *Membrii structurii* ***MEMORYSTATUS***.

Aplicațiile pot utiliza funcția *GlobalMemoryStatus* pentru determinarea cantității de memorie ce poate fi alocată fără un impact sever asupra altor programe. Informația returnată este volatilă, fără să existe nici o garanție că apelul succesiv la funcția *GlobalMemoryStatus* va returna aceeași informație. Pentru a înțelege mai bine cum funcționează *GlobalMemoryStatus* parcurgeți programul *Global_Mem_Status.cpp* de pe CD-ROM-ul atașat. Programul verifică starea curentă a memoriei și returnează cantitatea de memorie ferestrei programului.

1368 CREAREA UNUI HEAP ÎNTR-UN PROCES



În secțiunile precedente ați învățat modul de alocare în programe a memoriei din zona heap globală. Programul va aloca, de asemenea, blocuri mai mici de memorie din zona heap locală (privată). Funcția *HeapCreate* produce un obiect heap care poate fi utilizat de

procesul apelant. Funcția rezervă un bloc contiguu în spațiul de adrese virtuale ale procesului și alocă memorie fizică pentru o porțiune inițială, specificată, a blocului rezervat. Prototipul funcției *HeapCreate* este prezentat mai jos:

```
HANDLE HeapCreate(
DWORD flOptions, // indicator de alocare in heap
DWORD dwInitialSize, // dimensiune initiala a zonei heap
DWORD dwMaximumSize // dimensiune maxima a zonei heap
);
```

Parametrul *flOptions* specifică atributele opționale (indicatoare) ale noului heap. Aceste indicatoare afectează accesul următor la noul heap cu ajutorul funcțiilor *(HeapAlloc, HeapFree, HeapReAlloc* și *HeapSize)*. Puteți specifica unul sau mai multe din valorile prezentate în Tabelul 1368.

Valoare	Semnificație
HEAP_GENERATE_EXCEPTIONS	Specifică faptul că sistemul va semnala o eroare pentru a anunța eșuarea unei funcții, cum ar fi o condiție „out-of-memory" (lipsă memorie), în loc de a returna NULL.
HEAP_NO_SERIALIZE	Specifică faptul că excluderea reciprocă nu va fi utilizată când funcțiile de heap alocă memorie liberă din acest heap. Valoarea implicită, atunci când *HEAP_NO_SERIALIZE* nu este specificat, este serializarea accesului la heap. Serializarea unui heap permite ca două sau mai multe fire să aloce simultan și să elibereze memorie din heap.

Tabelul 1368 *Valorile acceptate ale parametrului* ***flOptions***.

Parametrul *dwInitialSize* specifică dimensiunea inițială, în octeți, a zonei heap. Această valoare determină cantitatea inițială de stocare fizică alocată pentru heap. Valoarea este rotunjită până la limita următoarei pagini. Pentru determinarea dimensiunii paginii pe calculatorul gazdă, se utilizează funcția *GetSystemInfo*. Dacă parametrul *dwMaximumSize* este diferit de zero, el specifică dimensiunea, în octeți, a zonei heap. Funcția *HeapCreate* rotunjește *dwMaximumSize* până la limita următoarei pagini, apoi rezervă un bloc de acea mărime în spațiul de adresare virtuală a procesului pentru heap. Dacă solicitările de alocare operate de funcțiile *HeapAlloc* sau *HeapReAlloc* depășesc cantitatea inițială de stocare fizică specificată de *dwInitialSize*, sistemul alocă pagini suplimentare de stocare fizică pentru heap, până la dimensiunea maximă a zonei heap.

În plus, dacă *dwMaximumSize* este diferit de zero, zona heap nu poate crește, intervenind o limitare absolută: dimensiunea maximă a blocului de memorie din heap este puțin mai mică decât 0x7FFF8 octeți (dimensiunea spațiului de adresă privat al procesului). Solicitările de alocare a unor blocuri mai mari de memorie vor eșua, chiar dacă dimensiunea maximă a zonei heap este suficient de mare pentru a conține blocul. Dacă *dwMaximumSize* este zero, el precizează că zona heap poate crește. Dimensiunea zonei heap este limitată numai de memoria disponibilă. Solicitările de alocare de blocuri mai mari de 0x7FFF8 octeți nu vor eșua automat; sistemul apelează funcția *VirtualAlloc* pentru a obține memoria necesară acestor blocuri mari. Aplicațiile care necesită alocări de blocuri mari, trebuie să fixeze *dwMaximumSize* la zero.

Dacă funcţia reuşeşte, valoarea returnată va fi identificatorul noului heap creat. Dacă funcţia eşuează, valoarea returnată va fi NULL. Pentru informaţii suplimentare despre erori, apelaţi funcţia *GetLastError*.

Funcţia *HeapCreate* produce un obiect heap privat din care procesul apelant poate aloca blocuri de memorie prin utilizarea funcţiei *HeapAlloc*. Dimensiunea iniţială determină numărul paginilor angajate, iniţial alocate pentru heap. Dimensiunea maximă determină numărul total de pagini rezervate. Aceste pagini angajate şi paginile rezervate produc un bloc contiguu în spaţiul de adresare virtuală a procesului în care zona heap poate creşte. Dacă solicitările făcute de *HeapAlloc* depăşesc dimensiunea curentă a paginilor angajate, sunt automat angajate pagini suplimentare din paginile rezervate, presupunând că este disponibilă stocarea fizică.

Memoria unui obiect heap privat este accesibilă numai procesului care a creat-o. Dacă o bibliotecă cu legare dinamică (DLL) produce un heap privat, acel heap este creat în spaţiul de adresare al procesului care a apelat biblioteca DLL şi este accesibil numai acelui proces.

Sistemul utilizează memorie din zona heap privată pentru stocarea structurilor de heap, astfel că nu întreaga dimensiune specificată a zonei heap este disponibilă în proces. De exemplu, dacă funcţia *HeapAlloc* solicită 64 de kiliocteţi (K) dintr-un heap cu dimensiune maximă de 64K, solicitarea poate eşua datorită supraîncărcării sistemului.

Dacă indicatorul *HEAP_NO_SERIALIZE* nu este specificat (situaţia implicită), zona heap va serializa accesul în cadrul procesului apelant. Serializarea asigură excluderea mutuală când două sau mai multe fire de execuţie încearcă să aloce sau să elibereze simultan blocuri din acelaşi heap. Serializarea scade mai puţin performanţa (adică Windows necesită mai mult timp de prelucrare), dar ea trebuie utilizată de fiecare dată când mai multe fire alocă sau eliberează memorie din acelaşi heap.

Stabilirea indicatorului *HEAP_NO_SERIALIZE* elimină excluderea reciprocă din heap. Fără serializare, două sau mai multe fire de execuţie care folosesc acelaşi identificator de heap pot încerca alocarea sau eliberarea simultană a memoriei, putând cauza coruperea în zona heap. Indicatorul *HEAP_NO_SERIALIZE* poate fi, în consecinţă, utilizat în siguranţă numai în următoarele situaţii:

- Procesul are un singur fir de execuţie.
- Procesul are fire multiple, dar numai un singur fir apelează funcţiile de heap pentru un anumit heap.
- Procesul are fire multiple şi aplicaţia oferă propriul mecanism pentru excluderea reciprocă într-un anumit heap.

1369 UTILIZAREA FUNCŢIILOR HEAP PENTRU GESTIONAREA PROCESELOR SPECIFICE DE MEMORIE



Aşa cum aţi învăţat, programele dumneavoastră trebuie să aloce cantităţi mici de memorie din heap cerute de un proces. Funcţia utilizată în general pentru alocarea unei astfel de memorii este *HeapAlloc*. Funcţia *HeapAlloc* alocă un bloc de memorie dintr-un heap. Memoria alocată cu *HeapAlloc* nu este deplasabilă. Prototipul funcţiei HeapAlloc este:

```
LPVOID HeapAlloc(
  HANDLE hHeap, // identificator pentru blocul de heap privat
  DWORD dwFlags,// indicatoare de control pentru alocarea
                // zonei heap
  DWORD dwBytes // numar de octeti de alocat
);
```

Parametrul *hHeap* specifică zona heap de la care va fi alocată memoria. Acest parametru este un identificator returnat de funcțiile *HeapCreate* sau *GetProcessHeap*. Parametrul *dwFlags* specifică multe aspecte controlabile ale alocării din heap. Specificarea oricăruia din aceste indicatoare va suprascrie indicatorul corespunzător în cazul în care zona heap a fost creată cu *HeapCreate*. Puteți specifica unul sau mai multe din valorile prezentate în Tabelul 1369.1

Fanion	Semnificație
HEAP_GENERATE_EXCEPTIONS	Specifică faptul că sistemul de operare va semnala o eroare pentru indicarea unui eșec al unei funcții, cum ar fi o condiție *out-of-memory*, în loc de a returna NULL.
HEAP_NO_SERIALIZE	Specifică faptul că excluderea reciprocă nu va fi utilizată în timpul accesării zonei heap de către funcția *HeapCreate*.
HEAP_ZERO_MEMORY	Specifică faptul că memoria alocată va fi inițializată de Windows cu zero.

Tabelul 1369.1 *Valorile parametrului **dwFlags**.*

Parametrul *dwBytes* specifică numărul de octeți pentru alocare. Dacă zona heap specificată de parametrul *hHeap* este un heap ce nu poate crește, *dwBytes* trebuie să fie mai mic decât 0x7FFF8. Un heap ce nu poate fi mărit se produce prin apelarea funcției *HeapCreate* cu o valoare diferită de zero. Dacă funcția reușește, valoarea returnată va fi un pointer la blocul de memorie alocat. Dacă funcția eșuează și ați specificat *HEAP_GENERATE_EXCEPTIONS*, valoarea returnată va fi NULL. Dacă funcția eșuează și ați specificat *HEAP_GENERATE_EXCEPTIONS*, funcția va genera excepțiile prezentate în Tabelul 1369.2.

Valoare	Semnificație
STATUS_NO_MEMORY	Încercarea de alocare a eșuat datorită lipsei de memorie disponibilă sau coruperii zonei heap.
STATUS_ACCESS_VIOLATION	Încercarea de alocare a eșuat datorită coruperii zonei heap sau a parametrilor necorespunzători.

Tabelul 1369.2 *Valorile de eroare pentru alocările eronate din heap.*

Coruperea zonei heap poate conduce la oricare din erorile arătate. Totul depinde de natura coruperii. Dacă funcția *HeapAlloc* reușește, ea va aloca cel puțin cantitatea de memorie cerută de programul apelant. Când cantitatea alocată în realitate este mai mare decât cantitatea cerută, procesul poate utiliza întreaga cantitate. Pentru a determina dimensiunea reală a blocului alocat, utilizați funcția *HeapSize*.

Pentru a elibera un bloc de memorie alocat de *HeapAlloc*, utilizați funcția *HeapFree*. Memoria alocată de *HeapAlloc* nu este deplasabilă. Datorită acestui fapt, este posibil ca zona heap să se fragmenteze. Rețineți că în cazul în care *HEAP_ZERO_MEMORY* nu este specificat, memoria alocată nu va fi inițializată cu zero.

Pentru a înțelege mai bune procesul efectuat de *HeapAlloc*, să analizăm programul *Heap_Strings.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Heap_String* creează un heap, apoi utilizează *HeapAlloc* pentru a aloca memorie din heap, pe care programul o tratează ca pe o matrice dinamică de șiruri de caractere. Când utilizatorul selectează din meniu opțiunea *Allocate!*, programul alocă memorie din heap pentru a stoca un nou șir și declararea unui pointer la memoria pe care alocarea precedentă a adăugat-o matricei. Dacă nu mai este loc disponibil în matrice, programul va utiliza funcția *HeapReAlloc* pentru extinderea matricei. Când utilizatorul selectează opțiunea *Free!* programul eliberează memoria pentru ultimul șir alocat. Când programul detectează că utilizatorul a eliberat suficientă memorie pentru a pune la dispoziție un spațiu neutilizat suficient, programul realocă zona heap pentru reducerea dimensiunii matricei. Mai mult, de fiecare dată când programul *Heap_Strings.cpp* realocă zona heap, el utilizează funcția *HeapCompact* pentru a o condensa.

1370 TESTAREA DIMENSIUNII MEMORIE ALOCATE DIN HEAP



Așa cum ați învățat, programele create de dumneavoastră în Windows vor aloca deseori cantități mici de memorie locală dintr-un heap privat. În secțiunile precedente ați creat un heap și ați alocat memorie din el. Programele dumneavoastră pot, de asemenea, utiliza funcții cum ar fi *HeapReAlloc* pentru a realoca spațiu dintr-un heap și *HeapFree* pentru a elibera memoria alocată din heap. În plus, programele dumneavoastră trebuie mereu să utilizeze funcția *HeapDestroy* pentru a distruge zonele heap private create. De multe ori însă, pe parcursul executării programelor, puteți să testați dimensiunea alocării din heap. Programul poate utiliza funcția *HeapSize* pentru a testa dimensiunea unui bloc de memorie alocat din heap. Funcția *HeapSize* returnează dimensiunea în octeți a blocului de memorie alocat din heap de către funcțiile *HeapAlloc* sau *HeapReAlloc*. Prototipul funcției *HeapSize* este următorul:

```
DWORD HeapSize(
    HANDLE hHeap,      // identificator pentru heap
    DWORD dwFlags,     // indicatoare de control al dimensiunii
                       // zonei heap
    LPCVOID lpMem      // pointer la memoria pentru care se
                       // returneaza dimensiunea
);
```

Parametrul *hHeap* specifică zona heap în care rezidă blocul de memorie. Acest indicator este returnat de funcția *HeapCreate* sau *GetProcessHeap*. Parametrul *dwFlags* specifică mai multe aspecte controlabile ale accesării blocului de memorie. Numai *HEAP_NO_SERIALIZE* este definit în prezent; celelalte valori sunt rezervate pentru utilizări viitoare. Specificarea lui *HEAP_NO_SERIALIZE* va suprascrie indicatorul corespunzător indicat de parametrul *flOptions* la crearea zonei heap cu funcția *HeapCreate*. Parametrul *lpMem* indică blocul de memorie a cărui dimensiune se va obține. Acesta este un pointer returnat de funcțiile *HeapAlloc* sau *HeapReAlloc*.

Dacă funcția reușește, valoarea returnată va fi dimensiunea, în octeți a blocului de memorie alocat. Dacă funcția eșuează, valoarea returnată de *HeapSize* va fi *0xFFFFFFFF*.

Pentru a înțelege mai bine prelucrările efectuate de funcția *HeapSize*, să analizăm programul *Heap_Size.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Heap_Size.cpp* produce un heap și alocă un bloc de memorie ce conține zero de 20 de octeți lungime. Programul *Heap_Size.cpp* apelează apoi funcția *HeapSize* pentru a afișa dimensiunea blocului alocat.

ALOCAREA UNUI BLOC DE MEMORIE VIRTUALĂ

C/C++1371

Așa cum ați învățat, programele dumneavoastră Windows vor aloca memorie utilizând unul din cele trei moduri de alocare: alocare din memoria heap globală, alocare din memoria heap privată și alocare directă de memorie virtuală. Așa cum ați învățat, utilizarea memoriei virtuale furnizează programului dumneavoastră control și opțiuni suplimentare pentru procesul de alocare. Totuși, spre deosebire de alte tipuri de alocare, veți efectua alocări de memorie cu o funcție *alloc*. Funcția *VirtualAlloc* diferă de alte funcții de alocare prin aceea că ea rezervă sau angajează o regiune de pagini în spațiul de adresare virtuală al procesului apelant. Windows inițializează în mod automat cu zero memoria alocată de programele dumneavoastră cu ajutorul funcției *VirtualAlloc*. Prototipul funcției *VirtualAlloc* este prezentat mai jos:

```
LPVOID VirtualAlloc(
    LPVOID lpAddress, // adresa regiunii pentru rezervare sau
                      // angajare
    DWORD dwSize,     // dimensiunea regiunii
    DWORD flAllocationType, // tipul alocarii
    DWORD flProtect   // tipul protectiei la acces
);
```

Funcția *VirtualAlloc* acceptă parametrii din Tabelul 1371.1.

Parametru	Descriere
lpAddress	Specifică adresa de început a regiunii ce va fi alocată. Dacă programul rezervă memorie, adresa specificată este rotunjită în jos la următoarea limită de 64Kb. Dacă memoria este deja rezervată și apelează acum *VirtualAlloc* pentru a angaja memoria, adresa este rotunjită în jos la următoarea limită de pagină. Pentru a determina dimensiunea paginii pe calculatorul gazdă, utilizați funcția *GetSystemInfo*. Dacă parametrul este NULL, sistemul determină unde va aloca regiunea.
dwSize	Specifică dimensiunea, în octeți, a regiunii. Dacă parametrul *lpAddress* este NULL, această valoare este rotunjită în sus la limita următoarei pagini. Altfel, paginile alocate cuprind toate paginile care conțin unul sau mai mulți octeți în intervalul de la *lpAddress* la *(lpAddress+dwSize)*. Aceasta înseamnă că intervalul de 2 octeți de la limita a două pagini are ca efect cuprinderea ambelor pagini în regiunea alocată.
flAllocationType	Specifică tipul de alocare. Puteți specifica orice combinație a valorilor prezentate în Tabelul 1371.2.

(continuare)

Parametru	Descriere
flProtect	Specifică tipul protecției la acces. Dacă folosiți *VirtualAlloc* pentru a angaja paginile, oricare din valorile prezentate în Tabelul 1371.3 pot fi specificate împreună cu indicatoarele de modificare a protecției *PAGE_GUARD* și *PAGE_NOCACHE.*

Tabelul 1371.1 *Parametrii funcției* ***VirtualAlloc***.

Așa cum se expune în Tabelul 1371.1, parametrul *flAllocationType* este utilizat pentru controlul alocării de memorie cu *VirtaulAlloc.* Tabelul 1371.2 specifică valorile care pot fi combinate pentru controlul alocării de memorie virtuală.

Fanion	Semnificație
MEM_COMMIT	Alocă stocare fizică în memorie sau în fișierul de paginare de pe disc pentru regiunea de pagini specificată. Cu alte cuvinte, protejează o parte din spațiul de adresare virtuală față de alte apeluri de alocare în cadrul aceluiași proces. Încercarea de a angaja o pagină deja angajată nu va avea ca efect nereușita funcției. Aceasta înseamnă că un interval de pagini angajate sau dezangajate, pot fi angajate fără a exista pericolul unei eșuări.
MEM_RESERVE	Rezervă un interval din spațiul de adresare virtuală a procesului fără să aloce nici o stocare fizică. Intervalul rezervat nu poate fi utilizat de către oricare alte operațiuni de alocare (funcția *malloc* sau *GlobalAlloc* etc.) decât după ce a fost eliberat. Paginile de rezervă pot fi angajate în apeluri ulterioare la funcția *VirtualAlloc.*
MEM_TOP_DOWN	Alocă memorie la cea mai înaltă adresă posibilă.

Tabelul 1371.2 *Tipurile posibile de alocare pentru funcția* ***VirtualAlloc***.

Datorită naturii alocării de pagini virtuale, puteți controla accesul la paginile virtuale pe care le angajați cu funcția *VirtualAlloc.* Așa cum se vede din Tabelul 1371.1, nu puteți specifica decât un singur tip de securitate a paginii, împreună cu modificatorii *PAGE_GUARD* și *PAGE_NOCACHE.* Tabelul 1371.3 prezintă indicatoarele de securitate pentru alocările de memorie virtuală.

Indicator	Semnificație
PAGE_READONLY	Activează numai dreptul la citire regiunii de pagini angajate. Încercarea de scriere în paginile doar cu acces la citire va duce la o violare de acces. Dacă sistemul diferențiază între dreptul de acces read-only și accesul la execuție, încercarea de a executa cod în regiunea angajată va duce la o violare de acces.
PAGE_READWRITE	Activează ambele drepturi de acces la scriere și citire pentru regiunea de pagini angajate.
PAGE_EXECUTE	Activează doar accesul la execuție regiunii de pagini angajate. Încercarea de a citi sau scrie în paginile cu acces doar la execuție va duce la o violare de acces.
PAGE_EXECUTE_READ	Activează drepturile de acces la execuție și scriere pentru regiunea de pagini angajate.

Indicator	Semnificație
PAGE_EXECUTE_READWRITE	Activează drepturile de acces la execuție, scriere și citire pentru regiunea de pagini angajate.
PAGE_GUARD	Paginile din regiune devin pagini de gardă. Încercarea de a citi sau scrie o pagină de gardă, face ca sistemul de operare să semnaleze excepția *STATUS_GUARD_PAGE* și să oprească starea de pagină de gardă. Astfel, paginile de gardă se comportă ca o alarmă la o singură tentativă de acces. Indicatorul PAGE_GUARD este un modificator de protecție a paginii. Aplicația îl utilizează cu unul sau mai multe alte indicatoare de protecție a paginii, cu o singură excepție: nu poate fi folosită cu *PAGE_NOACCESS*. Când o încercare eșuată de acces la scriere sau citire determină ca sistemul de operare să oprească starea de pagină de gardă, protecția paginii nu mai are loc. Dacă are loc o excepție a paginii de gardă pe parcursul unui serviciu de sistem, serviciul returnează, de regulă, un indicator de stare de eroare. Secțiunea 1372 explică în detaliu paginile de gardă.
PAGE_NOACCESS	Dezactivează toate accesele la regiunea paginilor angajate. Încercarea de a citi, scrie sau executa în paginile cu acces dezactivat va duce la o eroare de violare de acces, denumită *general protection fault (GP)*.
PAGE_NOCACHE	Nu permite depozitarea în *cache* a paginilor alocate anterior cu MEM_COMMIT. Atributele hardware ale memoriei fizice trebuie precizate cu „no cache". Microsoft nu recomandă utilizarea acestui indicator pentru uz general. El este util pentru driverele de dispozitiv (de exemplu, maparea unui buffer cadru video) fără depozitare în cache. Acest indicator este un modificator de protecție a paginii, valid numai când este utilizat cu una din facilitățile de protecție a paginii, alta decât *PAGE_NOACCESS*.

Tabelul 1371.3 *Indicatoarele de securitate acceptate pentru alocarea virtuală de pagini.*

Dacă funcția *VirtualAlloc* reușește, valoarea returnată este adresa de bază a regiunii de pagini alocată. Dacă funcția *VirtualAlloc* eșuează, valoarea returnată este NULL.

Funcția *VirtualAlloc* poate efectua următoarele operații:

- Angajarea unei regiuni de pagini rezervate de un apel anterior la funcția *VirtualAlloc*
- Rezervarea unei regiuni de pagini libere
- Rezervarea și angajarea unei regiuni de pagini libere

Puteți folosi *VirtualAlloc* pentru a rezerva un bloc de pagini și apoi să faceți apeluri suplimentare la *VirtualAlloc* pentru angajarea paginilor individuale dintr-un bloc rezervat. Rezervarea unui bloc de pagini permite unui anumit proces să rezerve un interval din spațiul său de adresare virtuală fără consumarea stocării de memorie fizică, până când acesta va fi necesar.

Fiecare pagină din spațiul de adresare virtuală a procesului va avea una din stările menționate în Tabelul 1371.4.

Stare	Semnificație
Liberă (free)	Pagina nu este angajată sau rezervată și nu este accesibilă procesului. Funcția *VirtualAlloc* poate rezerva, sau poate rezerva și angaja simultan, o pagină liberă.
Rezervată (reserved)	Intervalul de adrese nu poate fi utilizat de către alte funcții de alocare, dar pagina nu este accesibilă și nu deține stocare fizică asociată. *VirtualAlloc* poate angaja o pagină rezervată, dar nu o poate rezerva a doua oară. Funcția *VirtualFree* poate elibera o pagină rezervată, conferindu-i starea de pagină liberă.
Angajată (Committed)	Windows a alocat pagina pentru stocare, iar accesul este controlat de codul de protecție. Sistemul inițializează și încarcă fiecare pagină angajată în memoria fizică numai la prima încercare pentru citire sau scriere în pagina respectivă. Când procesul se încheie, sistemul eliberează stocarea pentru pagini angajate. *VirtualAlloc* poate angaja o pagină deja angajată. Aceasta înseamnă că puteți angaja un interval de pagini, indiferent dacă ele au fost deja angajate, fără ca funcția să eșueze. *VirtualFree* poate dezangaja o pagină angajată, elibera stocarea unei pagini sau poate simultan dezangaja și elibera o pagină angajată.

Tabelul 1371.4 *Stările posibile ale memoriei virtuale.*

Dacă parametrul *lpAddress* nu este NULL, funcția utilizează parametrii *lpAddress* și *dwSize* pentru a calcula regiunea de pagini ce urmează a fi alocată de *VirtualAlloc*. Starea curentă a întregului interval de pagini trebuie să fie compatibilă cu tipul de alocare specificat de parametrul *flAllocationType*. Altfel, funcția eșuează și nici una din pagini nu este alocată. Această cerință de compatibilitate nu previne angajarea unei pagini deja angajate (vezi Tabelul 1371.4).

Pentru a înțelege mai bine prelucrările efectuate de *VirtualAlloc*, să analizăm programul *Virtual_Allocate.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Virtual_Allocate.cpp* rezervă 1Mb de memorie virtuală atunci când programul trimite mesajul *WM_CREATE*. Când utilizatorul selectează articolul de meniu *Test*, programul angajează și utilizează 70Kb de memorie virtuală. Mai întâi, programul plasează valori în fiecare bloc de 1Kb de memorie alocată. În al doilea rând, programul schimbă dreptul de acces al întregului bloc de memorie angajată la *read-only*. În al treilea rând, programul accesează o valoare din memorie și o afișează într-o casetă de mesaj. În final, programul încearcă să fixeze o valoare în memorie, care are ca efect o eroare de protecție. Programul *Virtual_Allocate.cpp* utilizează un bloc *try-catch* pentru a trata eroarea de protecție.

1372 PAGINILE DE GARDĂ



Așa cum ați învățat în secțiunea 1371, o aplicație stabilește indicatorul *PAGE_GUARD* al modificatorului de protecție a paginii de memorie, în vederea stabilirii unei pagini de gardă. Puteți specifica acest indicator în funcțiile *VirtualAlloc*, *VirtualProtect* și *VirtualProtectEx*, împreună cu alte indicatoare de protecție a paginii. Puteți utiliza indicatorul *PAGE_GUARD* cu oricare alt indicator de protecție, cu excepția indicatorului *NO_ACCESS*.

Dacă programul încearcă să acceseze o adresă într-o pagină de gardă, sistemul de operare va semnala excepția *STATUS_GUARD_PAGE (0x80000001)*. Sistemul de operare va șterge

indicatorul *PAGE_GUARD*, ridicând paginii de memorie starea de pagină de gardă. Sistemul nu va opri următoarea încercare de acces la pagina de memorie cu o excepție *STATUS_GUARD_PAGE*.

Dacă excepția paginii de gardă apare în timpul unui serviciu de sistem, serviciul va eșua și va returna, de regulă, un indicator de stare de eroare. Deoarece sistemul ridică și starea de pagină de gardă acordată paginii de memorie, invocarea următoare a aceluiași serviciu de sistem nu va eșua din cauza unei excepții *STATUS_GUARD_PAGE* (dacă, desigur, cineva nu restabilește pagina de gardă).

În consecință, pagina de gardă se comportă ca o alarmă la o singură tentativă de acces la pagina de memorie. Acest fapt se poate dovedi util pentru o aplicație care trebuie să monitorizeze creșterea unor structuri dinamice mari de date. De exemplu, unele sisteme de operare, utilizează paginile de gardă pentru implementarea testărilor automate de stivă.

CD-ROM-ul care însoțește cartea de față cuprinde programul *Guard_Page.cpp*, care ilustrează comportamentul de alarmă al paginilor de gardă la o singură tentativă de acces. De asemenea, demonstrează cum poate eșua un serviciu de sistem (programul generează ieșiri la o fereastră DOS). După compilarea și executarea programului *Guard_Page.cpp*, ieșirea la monitor va arăta astfel:

```
Commited 512 bytes at address 003P0000
Cannot lock at 003P0000, error = 0x80000001
Second lock Achieved at 003P0000
C:\>
```

Rețineți că prima încercare de blocare a memoriei eșuează, semnalând excepția *STATUS_GUARD_PAGE*. A doua încercare reușește, deoarece prima încercare a oprit protecția de pagină de gardă a blocului de memorie.

BLOCURILE DE MEMORIE VIRTUALĂ

C/C++1373

Așa cum ați învățat, memoria virtuală vă oferă mijloace suplimentare și eficiente de alocare a unor blocuri mari de memorie. Totuși, o ilustrare a beneficiilor pentru programare aduse de memoria virtuală este de multe ori dificilă. Unul din cele mai ușoare moduri de a vedea beneficiile memoriei virtuale este considerarea unei matrice de mari dimensiuni a unei structuri complexe, de pildă a unei structuri pentru calcul tabelar. În cazul unei matrice bidimensionale, definiția ar fi asemănătoare celei de mai jos:

```
Cell TABLOUMARE[200][256];
```

Dacă dimensiunea structurii *Cell* ar fi de 128 de octeți, ea ar necesita 6.533.600 de octeți (200 x 256 x 128) de stocare fizică. Este clar că e vorba de o importantă cantitate de memorie fizică de alocat foii de calcul tabelar (majoritatea foilor de calcul tabelar nu vor folosi nici pe departe atâtea celule).

Ați mai putea utiliza o listă înlănțuită pentru a crea o structură de tip calcul tabelar. Prin abordarea listei înlănțuite, nu e nevoie decât că creați structurile *Cell* pentru celulele foii de calcul care conțin în realitate date. Deoarece majoritatea celulelor unei foi de calcul tabelar rămân neutilizate, metoda listei înlănțuite economisește o cantitate însemnată de memorie. Însă, utilizarea listelor înlănțuite face dificilă obținerea conținutului celulelor. De exemplu, dacă programul dumneavoastră solicită aflarea conținutului celulei din rândul 5, coloana 10, trebuie mai întâi să parcurgă lista pentru regăsirea celulei, ceea ce încetinește prelucrarea.

Memoria virtuală oferă un compromis între declararea matricei bidimensionale mai întâi și implementarea listelor înlănțuite extinse, de mare complexitate. Cu memoria virtuală obțineți un acces ușor și rapid la tehnica matriceală și la capacitatea superioară de stocare a listelor înlănțuite. Pentru a profita cel mai bine beneficiile metodei memoriei virtuale, programul dumneavoastră trebuie:

1. Să rezerve o regiune suficient de mare pentru a cuprinde întreaga matrice de structuri *Cell*. Așa cum ați învățat, rezervarea unei regiuni nu utilizează memorie fizică.
2. Să localizați adresa de memorie în regiunea rezervată unde trebuie să meargă structura *Cell* când utilizatorul introduce date în celulă.
3. Să angajați suficientă stocare fizică la adresa de memorie (localizată la punctul 2) pentru o structură *Cell*.
4. Să stabiliți membrii noii structuri *Cell*.

După maparea stocării fizice la locația corespunzătoare, programul dumneavoastră poate accesa stocarea fără producerea unei violări de acces. În mod cert, tehnica memoriei virtuale este o importantă îmbunătățire față de alte tehnici, deoarece programul angajează stocarea fizică numai pe măsură ce utilizatorul introduce date în celulele foii de calcul tabelar. Întrucât majoritatea celulelor din foaia de calcul sunt goale, programul nu va utiliza cea mai mare parte a regiunii rezervate. Puteți utiliza alocarea de memorie virtuală pentru a oferi programelor dumneavoastră un acces rapid la un mare număr de membri fără multe din sacrificiile cerute de vechile modele de memorie.

1374 ELIBERAREA MEMORIEI VIRTUALE



Așa cum ați învățat, programul poate utiliza funcția *VirtualAlloc* pentru a rezerva sau angaja pagini de memorie. După ce ați rezervat sau angajat pagini de memorie virtuală, programele dumneavoastră pot utiliza funcția *VirtualFree* pentru eliberarea sau dezangajarea acestor pagini.

Funcția *VirtualFree* eliberează sau dezangajează (sau efectuează ambele operații) o regiune de pagini în spațiul virtual de adresare a procesului apelant. Prototipul funcției *VirtualFree* este redat mai jos:

```
BOOL VirtualFree(
    LPVOID lpAddress, // adresa regiunii cu pagini angajate
    DWORD dwSize, // dimensiunea regiunii
    DWORD dwFreeType // tipul operatie de eliberare a memoriei
);
```

Parametrul *lpAddress* indică adresa de bază a regiunii de pagini pentru eliberare. Dacă parametrul *dwFreeType* include indicatorul *MEM_RELEASE*, acest parametrul trebuie să fie adresa de bază returnată de funcția *VirtualAlloc* când a fost rezervată regiunea de pagini. Parametrul *dwSize* specifică dimensiunea, în octeți, a regiunii care va fi eliberată. Dacă parametrul *dwFreeType* include indicatorul *MEM_RELEASE*, parametrul *dwSize* trebuie să fie zero. Altfel, regiunea de pagini afectate va include toate paginile care conțin unul sau mai mulți octeți în intervalul cuprins între parametrul *lpAddress* și (*lpAddress+dwSize*). Aceasta înseamnă că intervalul de 2 octeți de la limita a două pagini are ca efect eliberarea ambelor pagini. Parametrul *dwFreeType* specifică tipul operației de eliberare a memoriei. La apelul funcției *VirtualFree* se va utiliza numai una dintre valorile înscrise în Tabelul 1374.

Indicator	Semnificaţie
MEM_DECOMMIT	Dezangajează regiunea specificată de pagini angajate. Încercarea de a dezangaja o pagină neangajată nu va avea ca efect eşuarea funcţiei. Aceasta înseamnă că un interval de pagini angajate sau dezangajate anterior, pot fi dezangajate fără eroare.
MEM_RELEASE	Eliberează regiunea specificată de pagini rezervate. Dacă este specificat acest indicator, parametrul *dwSize* trebuie să fie zero, altfel funcţia eşuează.

Tabelul 1374 *Valorile acceptate de parametrul* ***dwFreeType***.

Programele dumneavoastră pot utiliza funcţia *VirtualFree* pentru a efectua următoarele operaţii:

- Dezangajarea unei regiuni de pagini angajate sau neangajate
- Eliberarea unei regiuni de pagini rezervate
- Dezangajarea şi eliberarea unei regiuni de pagini angajate sau neangajate

Pentru eliberarea unei regiuni de pagini, întregul interval de pagini trebuie să fie în aceeaşi stare (toate rezervate sau toate angajate), iar întreaga regiune rezervată iniţial cu *VirtualAlloc* trebuie eliberată în acelaşi timp. Dacă numai o parte a paginilor din regiunea iniţial rezervată sunt angajate cu *VirtualAlloc*, trebuie să apelaţi mai întâi *VirtualFree* pentru dezangajarea paginilor angajate, iar apoi să apelaţi din nou *VirtualFree* pentru eliberarea întregului bloc.

Paginile eliberate cu *VirtualFree* devin pagini libere, disponibile pentru operaţiile de alocare ulterioare. Încercarea de a citi sau de a scrie aceste pagini libere, duce la o excepţie de violare a accesului. *VirtualFree* poate dezangaja o pagină neangajată; aceasta înseamnă că un interval de pagini angajate sau neangajate pot fi dezangajate fără eroare. Dezangajarea unei pagini eliberează stocarea fizică, fie în memorie, fie în fişierul de paginare de pe disc. Dacă o pagină este dezangajată, dar nu eliberată, starea paginii se schimbă într-o pagină rezervată şi poate fi din nou angajată de un apel ulterior al funcţiei *VirtualAlloc*. Încercarea de a citi sau scrie o pagină rezervată duce la o excepţie de violare a accesului.

Starea curentă a întregului interval de pagini trebuie să fie compatibilă cu tipul operaţiunii de eliberare specificat de parametrul *dwFreeType*. Altfel, funcţia eşuează şi nici o pagină nu este eliberată sau dezangajată.

GESTIONAREA PAGINILOR DE MEMORIE VIRTUALĂ

C/C++1375

Aşa cum aţi învăţat, programele dumneavoastră vor utiliza pagini de memorie virtuală, atât pentru a extinde accesul programului la memorie, cât şi pentru a face posibilă alocarea unor mari blocuri de memorie în spaţiul de memorie virtuală. Veţi solicita frecvent informaţii despre un interval de pagini în cadrul spaţiului de adresare virtuală a unui proces. Funcţia *VirtualQuery* furnizează informaţii despre un interval de pagini din spaţiul de adresare virtuală a unui proces apelant. Prototipul funcţiei este prezentat mai jos:

```
DWORD VirtualQuery(
    LPCVOID lpAddress, // adresa regiunii
    PMEMORY_BASIC_INFORMATION lpBuffer, // adresa bufferului de
                                        // informatii
```

```
    DWORD dwLength // dimensiunea bufferului
);
```

Funcția *VirtualQuery* acceptă cei trei parametri prezentați în Tabelul 1375

Parametru	Descriere
lpAddress	Indică adresa de bază a regiunii de pagini care fac obiectul interogării. Această valoare este rotunjită în jos la limita următoarei pagini. Pentru a afla dimensiunea unei pagini pe calculatorul gazdă, utilizați funcția *GetSystemInfo*.
lpBuffer	Indică o structură *MEMORY_BASIC_INFORMATION* în care este returnată informația despre intervalul specificat de pagini.
dwLength	Specifică dimensiunea, în octeți, a bufferului indicat de parametrul *lpBuffer*.

Tabelul 1375 *Parametrii funcției* ***VirtualQuery***.

Funcția *VirtualQuery* furnizează informații despre o regiune de pagini consecutive începând de la o adresă specificată, care au următoarele atribute:

- Starea tuturor paginilor este aceeași în cazul utilizării indicatoarelor *MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED* sau *MEM_IMAGE*.
- Dacă pagina inițială nu este liberă, toate paginile din regiune sunt parte a aceleiași alocări de pagini rezervate prin apelul la funcția *VirtualAlloc*.
- Accesul tuturor paginilor este același în cazul utilizării indicatoarelor *PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD* sau *PAGE_NOCACHE*.

Funcția *VirtualAlloc* determină atributele primei pagini din regiune și apoi scanează paginile următoare până la scanarea întregului interval de pagini sau până când întâlnește o pagină cu seturi de atribute necorespunzătoare. Funcția returnează atributele și dimensiunea, în octeți, a regiunii de pagini cu atribute corespunzătoare. De exemplu, dacă avem o regiune de 40Mb de memorie liberă și este apelată funcția *VirtualQuery* la o pagină de 10Mb din regiune, vom obține starea de *MEM_FREE* și dimensiunea de 30Mb.

Funcția *VirtualQuery* oferă informații despre regiunea de pagini din memoria procesului apelant, iar funcția *VirtualQueryEx* oferă informații despre regiunea de pagini din memoria procesului specificat. CD-ROM-ul care însoțește cartea de față, cuprinde programul *Virtual_Query.cpp*. Programul alocă mai întâi un bloc de 70Kb de memorie virtuală. Apoi programul invocă funcția *VirtualQuery* care returnează dimensiunea regiunii ocupată de memorie în acel moment. Observați că dimensiunea regiunii este divizibilă cu 4096 de octeți (4Kb), aceasta fiind dimensiunea paginii la calculatoarele x86.

1376 PROCESELE



Așa cum ați învățat, una dintre cele mai puternice caracteristici ale sistemului Windows este suportul multitasking, adică rularea simultană în memorie a mai multor procese. În următoarele treizeci de secțiuni, veți învăța mai mult despre gestionarea proceselor și a firelor de execuție, una dintre cele mai importante capacități ale sistemului Windows.

Un *proces* este obiectul care deține toate resursele unei aplicații. Un proces Windows poate crea unul sau mai multe fire de execuție. Un fir de execuție este o cale independentă de execuție în cadrul unui proces cu care firul partajează spațiul de adresă, codul și datele globale. Fiecare fir posedă propriul set de registre, propria stivă, propriile mecanisme de intrare, inclusiv o coadă privată de mesaje. Windows 95 și Windows NT alocă fracțiuni de timp pentru CPU într-un regim fir-cu-fir și efectuează activități multitasking de preempțiune (cu alte cuvinte, ordonează firele în funcție de prioritățile alocate fiecăruia).

În plus, un proces cuprinde alocări de memorie globală, pagini virtuale și așa mai departe. Figura 1376.1 arată un model logic de relații între fire și procese.

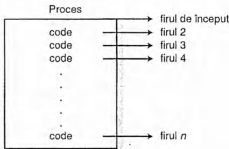


Figura 1376.1 *Relațiile dintre fire și procese.*

După cum ați învățat, Windows alocă memorie virtuală proceselor, în măsura în care au nevoie de ea. În consecință, când luați în considerare modelul de memorie al unui calculator pe care rulează simultan mai multe procese, este importantă identificarea procesului activ curent. Aflarea procesului activ curent este importantă deoarece Windows va atașa automat prioritate înaltă majorității solicitărilor CPU exercitate de procesul activ. Secțiunea 1400 discută în detaliu gestionarea de către sistemul de operare a priorității firelor de execuție. În plus, Windows distribuie de regulă memorie fizică suplimentară pentru accelerarea execuției procesului activ curent. Figura 1376.2 prezintă un exemplu de model de memorie pentru trei aplicații simultane, în care aplicația activă curent este *first_app.*

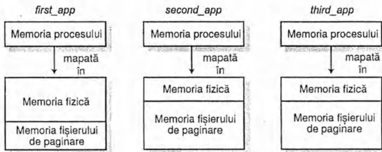


Figura 1376.2 *Un exemplu de model de memorie pentru trei aplicații care se execută*

Însă, în eventualitatea în care utilizatorul face activ programul *second_app*, Windows va realoca memorie fizică într-o manieră în care va elibera mai mult spaţiu pentru executarea programului *second_app*, cum se prezintă în figura 1376.3.

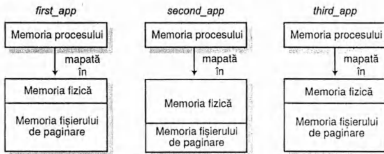


***Figura 1376.3** Un alt exemplu de model de memorie pentru trei aplicaţii care se execută simultan.*

În următoarele cinci secţiuni, veţi deprinde bazele creării şi gestionării proceselor. În secţiunile ulterioare veţi deprinde bazele creării şi gestionării firelor de execuţie. E important să înţelegeţi bine bazele proceselor (care reprezintă containere pentru fire) înainte de a începe lucrul cu firele de execuţie.

1377 CREAREA UNUI PROCES



Aşa cum aţi învăţat, de regulă, programele dumneavoastră vor rula în cadrul unui singur proces. Acest proces va conţine unul sau mai multe fire de execuţie. Funcţia *CreateProcess* produce un proces nou şi firul său de execuţie primar (denumit frecvent proces copil). Noul proces execută fişierul executabil specificat. *CreateProcess* permite procesului părinte (adică procesul care apelează funcţia *CreateProcess*) să specifice mediul de operare a noului proces, inclusiv directorul său de lucru, modul în care va apărea implicit pe ecran, variabilele de mediu şi prioritatea procesului. Windows transmite linia de comandă şi conţinutul ei către procesul copil. Funcţia *CreateProcess* deţine următorul prototip:

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName, // numele modulului executabil
    LPTSTR lpCommandLine, // sirul liniei de comanda
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // atributele de
                              // securitate ale procesului
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // atributele de
                              // securitate ale firului
    BOOL bInheritHandles, // identificator pentru indicatorul
                        // de mostenire
    DWORD dwCreationFlags, // indicatoare de creare
    LPVOID lpEnvironment, // pointer la noul bloc de mediu
    LPCTSTR lpCurrentDirectory, // pointer la numele
                              // directorului curent
```

```
    LPSTARTUPINFO lpStartupInfo, // pointer la STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation // pointer la
                                  // PROCESS_INFORMATION
);
```

Așa cum vedeți, funcția *CreateProcess* este puternic parametrizată. Tabelul 1377.1 expune parametrii funcției *CreateProcess*

Parametru	Descriere
lpApplicationName	Pointer la un șir terminat cu NULL care precizează modulul ce trebuie executat. Șirul poate specifica întreaga cale și numele de fișier cu modulul ce trebuie executat. Șirul poate specifica și un nume parțial. În acest caz, funcția utilizează unitatea curentă de disc și directorul curent pentru îndeplinirea specificației. Parametrul *lpApplicationName* poate fi NULL. În acest caz, numele modulului trebuie să fie primul simbol delimitat cu spații în șirul în *lpCommandLine*. Modulul specificat poate fi o aplicație Win32. El poate fi de alt tip (de exemplu, MS-DOS sau OS/2) dacă subsistemul corespunzător este disponibil pe calculator.
***Observație:** Sub Windows NT, dacă modulul executabil este o aplicație pe 16 biți, **lpApplicationName** trebuie să fie NULL, iar șirul spre care indică **lpCommandLine** trebuie să specifice modulul executabil.*	
lpCommandLine	Pointer la un șir terminat în NULL care specifică linia de comandă ce trebuie executată. Parametrul *lpCommandLine* poate fi NULL. În acest caz, funcția utilizează șirul spre care indică *lpApplicationName* ca linie de comandă. Dacă atât *lpApplicationName*, cât și *lpCommandLine* sunt ne-nule, *lpApplicationName* specifică modulul de executat, iar *lpCommandLine* specifică linia de comandă. Noul proces poate utiliza *GetCommandLine* pentru preluarea întregii linii de comandă. Procesele de tip run-time ale limbajului C pot utiliza argumentele *argc* și *argv*. Dacă *lpApplicationName* este NULL, primul simbol delimitat de spații specifică numele modulului. Dacă numele fișierului nu conține o extensie, Windows presupune că este .EXE. Dacă numele de fișier se termină cu un punct (.) fără extensie sau numele fișierului conține calea, extensia .EXE nu este adăugată. Dacă numele fișierului nu conține calea directorului, Windows caută fișierele executabile în următoarea succesiune: 1. Directorul din care s-a încărcat aplicația 2. Directorul curent al procesului părinte 3. Windows 95: directorul sistem al Windows. Utilizați funcția *GetSystemDirectory* pentru obținerea căii acestui director. 4. Windows NT: directorul sistem pe 32 biți din Windows. Utilizați funcția *GetSystemDirectory* pentru obținerea căii acestui director. Numele directorului este în general *SYSTEM32*. 5. Windows NT: directorul sistem pe 16 biți al Windows. Acest director este căutat deși nu există nici o funcție Win32 pentru obținerea căii de acces la acest director. Numele directorului este în general *SYSTEM*.

(continuare)

Parametru	Descriere
	6. Directorul Windows. Utilizați funcția *GetWindowsDirectory* pentru obținerea căii acestui director. Numele directorului este în general Windows. 7. Directoarele Windows enumerate în variabila de mediu PATH. Dacă procesul care va fi realizat cu *CreateProcess* este o aplicație MS-DOS sau o aplicație Windows, *lpCommandLine* trebuie să fie numele întreg al liniei de comandă în care primul element este numele aplicației. Deoarece *CreateProcess* lucrează bine și în aplicațiile Win32, ar trebui să stabiliți parametrul *lpCommandLine* la o linie de comandă completă și pentru programele Win32.
lpProcessAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care determină dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpProcessAttributes* este NULL, identificatorul nu poate fi moștenit.

***Observație:** Sub Windows NT, membrul **lpSecurityDescriptor** al structurii specifică descriptorul de securitate al noului proces. Dacă **lpProcessAttributes** este NULL, procesul preia descriptorul de securitate implicit. Sub Windows 95, membrul **lpSecurityDescriptor** este ignorat.*

Parametru	Descriere
lpThreadAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care determină dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpThreadAttributes* este NULL, identificatorul nu poate fi moștenit.

***Observație:** Sub Windows NT, membrul **lpSecurityDescriptor** al structurii specifică descriptorul de securitate al firului principal. Dacă **lpThreadAttributes** este NULL, firul preia descriptorul de securitate implicit. Sub Windows 95, membrul **lpThreadAttributes** este ignorat.*

Parametru	Descriere
bInheritHandles	Precizează dacă noul proces moștenește identificatorii de la procesele apelante. Dacă valoarea este TRUE, fiecare identificator deschis transmisibil din procesul apelant este moștenit de către noul proces. Identificatorii moșteniți au aceiași valoare și drepturi de acces ca și identificatorii inițiali.
dwCreationFlags	Specifică indicatoare suplimentare care controlează prioritatea clasei și crearea procesului. Pot fi specificate, în orice combinație, valorile prezentate în Tabelul 1377.2 (cu excepția celor special menționate). Parametrul *dwCreationFlags* mai controlează clasa de prioritate a noului proces, pe care o folosește Windows pentru a planifica prioritățile firelor procesului. Dacă Windows nu specifică nici unul din indicatoarele clasei de prioritate prezentate în Tabelul 1377.2, clasa de prioritate implicită este *NORMAL_PRIORITY_CLASS*. Dacă clasa de prioritate la crearea procesului este *IDLE_PRIORITY_CLASS*, clasa de prioritate implicită a procesului copil este, de asemenea, *IDLE_PRIORITY_CLASS*. Programele dumneavoastră pot specifica unul din indicatoarele de prioritate din Tabelul 1377.3.

Parametru	Descriere
lpEnvironment	Indică un bloc de mediu al noului proces. Dacă parametrul este NULL, noul proces utilizează mediul procesului apelant. Un bloc de mediu constă într-un bloc terminat cu NULL format din șiruri terminate cu NULL. Fiecare șir este prezentat în forma *nume = valoare*. Datorită faptului că semnul egal este utilizat ca separator, el nu trebuie utilizat în numele variabilei de mediu. Dacă o aplicație furnizează un bloc de mediu și nu transmite NULL pentru acest parametru, informațiile din directorul curent al unităților sistemului nu sunt automat repartizate noului proces. Un bloc de mediu poate conține caractere Unicode sau ANSI. Dacă blocul de mediu spre care indică *lpEnvironment,* conține caractere Unicode, Windows va stabili indicatorul *CREATE_UNICODE_ENVIRONMENT* în câmpul *dwCreationFlags.* Dacă blocul conține caractere ANSI, acel indicator va fi șters. Observați că un bloc de mediu ANSI este terminat cu doi octeți zero: unul pentru ultimul șir și unul pentru terminarea blocului. În plus, un bloc de mediu Unicode se încheie cu patru octeți zero: doi pentru ultimul șir și încă doi pentru încheierea blocului.
lpCurrentDirectory	Indică un șir de caractere terminat cu NULL care specifică unitatea și directorul curent al procesului copil. Șirul trebuie să indice calea de acces și numele complet de fișier, inclusiv litera unității. Dacă parametrul este NULL, noul proces este creat cu aceeași unitate și același director curent ca procesul apelant. Această opțiune este oferită de Windows în primul rând pentru programele shell care pun în execuție aplicații și care specifică unitatea inițială și directorul de lucru.
lpStartupInfo	Indică o structură *STARTUPINFO* care specifică modul în care va apărea fereastra principală a noului proces.
lpProcessInformation	Indică o structură *PROCESS_INFORMATION* care primește informații de identificare a noului proces.

Tabelul 1377.1** Parametrii funcției* ***CreateProcess.

Așa cum ați văzut în Tabelul 1377.1, parametrul *dwFlags* acceptă unul sau mai multe indicatoare de creare și o valoare pentru clasa de prioritate. Tabelul 1377.2 prezintă indicatoarele de creare acceptate de parametrul *dwFlags*.

Valoare	Semnificație
CREATE_DEFAULT_ERROR_MODE	Noul proces nu moștenește modul de eroare al procesului apelant. În schimb, *CreateProcess* conferă noului proces modul de eroare curent implicit. O aplicație stabilește modul curent implicit de eroare prin apelarea funcției *SetErrorMode*. Acest indicator este util în special pentru aplicațiile shell cu fire de execuție multiple care rulează în regim de dezactivare a erorilor de hardware. Comportamentul implicit al funcției *CreateProcess* este să facă posibilă moștenirea de către noul proces a procesului apelant. Stabilirea acestui indicator modifică acest comportament implicit.

(continuare)

Valoare	Semnificație
CREATE_NEW_CONSOLE	Noul proces moștenește o nouă consolă, în loc să moștenească consola părintelui. Acest indicator nu poate fi utilizat împreună cu *DETACHED_PROCESS*.
CREATE_NEW_PROCESS_GROUP	Noul proces este procesul rădăcină al unui nou grup de procese. Acest grup de procese cuprinde toate procesele descendente ale acestui proces rădăcină. Identificatorul de proces al noului grup de procese este același cu identificatorul procesului care este returnat în parametrul *lpProcessInformation*.
CREATE_SEPARATE_WOW_VDM	Numai pentru Windows NT: acest indicator este valid numai la începerea unei aplicații Windows pe 16 biți. Dacă stabiliți acest indicator, Windows rulează noul proces pe o mașină DOS virtuală (VDM) privată. În mod implicit, toate aplicațiile Windows pe 16 biți sunt rulate ca fire într-un singur VDM (Virtual DOS Machine) partajat. Avantajul rulării pe mașini VDM separate este acela că o cădere distruge numai un singur VDM, toate celelalte programe rulate în VDM-uri distincte continuă să funcționeze normal. De asemenea, aplicațiile Windows pe 16 biți rulate în VDM-uri separate dețin interogări de intrare separate. Aceasta înseamnă că în cazul în care o aplicație se oprește pentru un timp, aplicațiile din alte VDM-uri continuă să primească intrări.
CREATE_SHARED_WOW_VDM	Numai pentru Windows NT: indicatorul este valid numai la pornirea aplicației Windows pe 16 biți. Dacă opțiunea *DefaultSeparateVDM* din secțiunea Windows a fișierului WIN.INI este *TRUE*, acest indicator face ca funcția *CreateProcess* să suprascrie opțiunea și să ruleze noul proces în VDM-ul partajat.
CREATE_SUSPENDED	Firul primar al noului proces este creat într-o stare suspendată și nu rulează până când un alt fir (din alt proces) nu apelează funcția *ResumeThread*.
CREATE_UNICODE_ENVIRONMENT	Dacă este activ, blocul de mediu spre care indică *lpEnvironment* utilizează caractere Unicode. Dacă nu este activ, blocul de mediu utilizează caractere ANSI.
DEBUG_PROCESS	Dacă indicatorul este activ, procesul apelant este tratat ca depanator, iar noul proces este cel depanat de procesul apelant. Sistemul înștiințează depanatorul despre toate evenimentele de depanare intervenite în procesul depanat. Când creați un proces cu acest indicator activ, numai firul apelant (firul care a apelat *CreateProcess*) poate apela funcția *WaitForDebugEvent*.

Valoare	Semnificație
DEBUG_ONLY_THIS_PROCESS	Dacă nu este activ, iar Windows depanează la acel moment procesul apelant, noul proces devine un alt proces supus depanării de către depanatorul procesului apelant. Dacă procesul apelant nu este depanat de Windows, nu are loc nici o operație de depanare ca urmare a acestui indicator.
DETACHED_PROCESS	Pentru procesele de consolă, noul proces nu are acces la consola procesului părinte. Noul proces poate apela funcția *AllocConsole* mai târziu pentru a crea o nouă consolă. Acest indicator nu poate fi utilizat cu indicatorul *CREATE_NEW_CONSOLE*.

Tabelul 1377.2 *Valorile acceptate de parametrul* ***dwFalgs***.

Așa cum ați învățat, programele dumneavoastră pot selecta una sau mai multe indicatoare de creare pentru parametrul *dwFalgs*. Însă, programele dumneavoastră pot specifica un singur indicator de clasă de prioritate pentru parametrul *dwFlags*. Indicatorul clasei de prioritate va avea una din valorile prezentate în Tabelul 1377.3.

Indicator	Semnificație
HIGH_PRIORITY_CLASS	Indică un proces care efectuează activități ce solicită o executare imediată pentru a rula corect. Firele de execuție ale unui proces cu o clasă prioritate *mare* (high) au întâietate față de procesele cu clase de prioritate *normală* sau *mică (idle)*. Un exemplu este *Windows Task List* care trebuie să răspundă rapid când este apelată de utilizator, indiferent de încărcarea sistemului de operare. Fiți extrem de atenți când utilizați clasa de prioritate *mare*, pentru că o aplicație cu această clasă de prioritate poate utiliza aproape toate ciclurile disponibile de CPU.
IDLE_PRIORITY_CLASS	Indică un proces ale cărui fire rulează numai când sistemul este inactiv și dă întâietate firelor oricărui proces care rulează într-o clasă de prioritate superioară. Un exemplu este un program salvator de ecran. Clasa de prioritate *mică* este moștenită de procesele copil.
NORMAL_PRIORITY_CLASS	Indică un proces normal fără nici o cerință specială de programare a activității.
REALTIME_PRIORITY_CLASS	Indică un proces care prezintă cea mai înaltă prioritate posibilă. Firele din clasa de prioritate în timp real (*realtime*) au întâietate față de toate celelalte procese, inclusiv față de procesele sistemului de operare care efectuează operații importante. De exemplu, un proces în timp real, care rămâne în execuție mai mult decât un interval de timp foarte scurt, poate avea ca efect imposibilitatea golirii rezervelor cache pe disc sau absența răspunsurilor de la mouse.

Tabelul 1377.3 *Indicatoarele de prioritate acceptate de funcția* ***CreateProcess***.

Pe lângă crearea unui proces, *CreateProcess* produce, de asemenea, un nou obiect fir de execuție. Firul este creat cu o stivă inițială a cărei dimensiune este descrisă în antetul de imagine din fișierul executabil al programului. Firul începe execuția la punctul de intrare al

imaginii. Noul proces și identificatorii noului fir sunt creați cu drepturi depline de acces. Dacă nu este furnizat un descriptor de securitate, fiecare identificator poate fi utilizat în orice funcție care reclamă un identificator de obiect de acel tip. Dacă funcția oferă un descriptor de securitate, programul efectuează o testare de acces asupra tuturor utilizărilor ulterioare ale identificatorului înainte ca accesul să fie permis. Dacă testarea de acces nu permite accesul, procesul solicitant nu va fi capabil să utilizeze identificatorul pentru a avea acces la fir.

Windows atribuie procesului un identificator pe 32 de biți. Identificatorul este valid până la terminarea procesului. Identificatorul poate fi utilizat pentru identificarea procesului sau specificarea funcției *OpenProcess* pentru deschiderea unui identificator la proces. Windows mai alocă un identificator de fir de execuție de 32 de biți, pentru firul inițial din proces. Identificatorul este valid până la terminarea firului și poate fi utilizat pentru identificarea unică a firului din sistem. Acești identificatori sunt returnați în structura *PROCESS_INFORMATION*.

Când specificați numele unei aplicații în șirurile *lpApplicationName* sau *lpCommandLine*, nu are importanță dacă numele aplicației cuprinde extensia de fișier, cu o excepție: o aplicație MS-DOS sau Windows a cărei extensie de fișier este .COM trebuie să cuprindă extensia .COM. Firul apelant poate utiliza funcția *WaitForInputIdle* pentru a aștepta până când noul proces își termină inițializarea și așteaptă intrări de la utilizator, fără să existe intrări în așteptare. Acest procedeu se poate dovedi util pentru sincronizarea dintre procesele părinte și procesele copil, pentru că funcția *CreateProcess* returnează controlul fără să aștepte ca noul proces să-și încheie inițializarea. De exemplu, procesul care creează poate utiliza funcția *WaitForInputIdle*, înainte de a încerca să găsească o fereastră asociată noului proces.

Modul de terminare a unui proces recomandat de Microsoft este utilizarea funcției *ExitProcess*, pentru că această funcție anunță toate bibliotecile cu legare dinamică (DLL) atașate procesului despre terminarea sa iminentă. Alte mijloace de a termina un proces nu anunță bibliotecile DLL atașate. Observați că atunci când un fir apelează *ExitProcess*, celelalte fire ale procesului se încheie fără a avea ocazia să execute cod suplimentar (inclusiv codul de încheiere a firelor din bibliotecile DLL atașate).

Windows serializează între ele, în cadrul unui proces, funcțiile *ExitProcess, ExitThread, CreateThread, CreateRemoteThread* împreună cu procesul care începe să ruleze (ca rezultat al apelului la *CreateProcess*). Numai unul din aceste evenimente se poate petrece într-un spațiu de adresă, la un moment dat. Aceasta înseamnă aplicarea următoarelor restricții:

- Pe parcursul rutinelor de pornire a procesului și de inițializare a bibliotecilor DLL, pot fi create noi fire de execuție, dar ele nu încep execuția până când Windows nu încheie inițializarea bibliotecii DLL pentru proces.
- Numai un singur fir într-un proces poate fi prezent la un moment dat, în inițializarea unui DLL sau a unei rutine detașate.
- Funcția *ExitProcess* nu se returnează decât când nu mai există fire în inițializările de DLL sau în rutinele detașate.

Procesul creat rămâne în sistem până când toate firele din cadrul procesului s-au încheiat și toți identificatorii de proces și oricare din firele sale s-au închis în urma apelurilor la funcția *CloseHandle*. Atât identificatorii pentru proces, cât și cei pentru firele principale trebuie închise prin apeluri la funcția *CloseHandle*. Dacă acești identificatori nu mai sunt necesari, este bine să fie imediat închiși, după ce este creat procesul.

Când se încheie ultimul fir din proces, sunt lansate următoarele evenimente:

- Toate obiectele deschise de către proces sunt implicit închise.
- Starea de terminare a procesului (returnată de *GetExitCodeProcess*) se schimbă de la valoarea sa inițială *STILL_ACTIVE* la starea de terminare a ultimului fir.
- Windows stabilește obiectul fir al firului principal pe starea de semnalizare, satisfăcând orice fir aflat în așteptarea obiectului.
- Windows stabilește obiectul de proces pe starea de semnalizare, satisfăcând orice fir aflat în așteptarea obiectului.

Dacă directorul curent al unității C este \TEORA\C&C++, există o variabilă de mediu numită =C: a cărei valoare este C:\TEORA\C&C++. Așa cum ați observat în descrierea precedentă pentru *lpEnvironment*, informațiile despre directorul curent nu se propagă automat spre un nou proces atunci când parametrul *lpEnvironment* al funcției *CreateProcess* nu este NULL. Aplicația trebuie să transmită manual informațiile directorului curent către noul proces. Pentru a proceda astfel, aplicația trebuie să creeze explicit șiruri cu variabile de mediu =X, să le ordoneze alfabetic (pentru că Windows 95 și Windows NT folosesc un mediu sortat) apoi să le pună în blocul de mediu specificat de *lpEnvironment*. De regulă, după sortare, ele vor merge în fața blocului de mediu menționat.

O modalitate de a obține variabila directorului curent pentru o unitate X este apelul la funcția *GetFullPathName(„X:",. .)*. Aceasta permite aplicației să evite scanarea blocului de mediu. Dacă întreaga cale returnată este X:\, nu este nevoie să transmitem acea valoare ca date de mediu, pentru că directorul rădăcină este directorul curent implicit al unității X al noului proces. Funcția *CreateProcess* returnează identificatorul care are acces de tip *PROCESS_ALL_ACCESS* la obiectul proces.

Directorul curent specificat de parametrul *lpCurrentDirectory* este directorul curent al procesului copil. Directorul curent specificat la articolul în parametrul *lpCommandLine* este directorul curent al procesului părinte.

Observație: *În Windows NT, când un proces este creat cu specificarea identificatorului de prioritate* ***CREATE_NEW_PROCESS_GROUP****, un apel implicit la* ***SetConsoleCtrlHandler (NULL,True)*** *se face din partea noului proces; aceasta înseamnă că noul proces are semnalul CRTL+C dezactivat. Aceasta permite programelor shell bune să controleze ele însele CTRL+C și să transmită selectiv acest semnal către subprocese. CTRL+BREAK nu este dezactivat și poate fi utilizat pentru întreruperea procesului/grupului de proces.*

TERMINAREA PROCESELOR

C/C++1378

Așa cum ați învățat, programul dumneavoastră va fi executa în cadrul unui proces. În secțiunea 1377 ați învățat cum să creați un proces. La fel ca în cazul majorității operațiilor de alocare și creare efectuate de programul dumneavoastră, rămâne în responsabilitatea dumneavoastră să ieșiți din proces atunci când și-a încheiat funcțiunile. Funcția dedicată pentru închiderea executării unui proces curent este *ExitProcess*. Funcția *ExitProcess* încheie un proces împreună cu toate firele sale și returnează controlul către locația apelantă. Prototipul funcției *ExitProcess* este următorul:

```
VOID ExitProcess(
    UINT uExitCode // iese din proces cu toate firele
);
```

Parametrul *uExitCode* conține codul de ieșire pentru proces și pentru toate firele care terminate de proces, ca rezultat al apelului la *ExitProcess*. Utilizați funcția *GetExitCodeProcess* pentru preluarea valorii de ieșire a procesului. Utilizați funcția *GetExitCodeThread* pentru preluarea valorii de ieșire a firului de execuție.

Pentru închiderea unui proces trebuie întotdeauna apelată funcția *ExitProcess*. Această funcție oferă o închidere corectă a procesului. Aceasta include apelarea funcției de la punctul de intrare al tuturor bibliotecilor cu legare dinamică (DLL) atașate, cu o valoare care precizează că procesul se desprinde de DLL. Dacă procesul se încheie prin apelul la *TerminateProcess*, bibliotecile DLL la care este atașat procesul nu sunt notificate despre terminarea procesului.

După ce toate bibliotecile DLL atașate au executat valoarea de terminare a procesului, funcția *ExitProcess* termină efectiv procesul curent. Terminarea procesului are ca efect următoarele:

1. Sunt închiși toți identificatorii de obiect deschiși de proces.
2. Toate firele din cadrul procesului își termină execuția.
3. Starea obiectului din proces devine semnalizată, satisfăcând toate firele care s-au aflat în așteptarea terminării procesului.
4. Starea tuturor firelor din proces devine semnalizată, satisfăcând toate firele care s-au aflat în așteptarea terminării firelor procesului ce se încheie.
5. Starea de terminare a procesului se schimbă din valoarea *STILL_ACTIVE* la valoarea de ieșire (exit) a procesului.

Terminarea unui proces nu are ca efect terminarea proceselor copil. Terminarea unui proces nu duce în mod necesar la eliminarea din sistemul de operare a obiectului proces. Un proces este șters atunci când ultimul identificator la proces este închis. Windows serializează funcțiile *ExitProcess, ExitThread, CreateThread, CreateRemoveThread* împreună cu procesul care începe să ruleze (ca rezultat al apelului la *CreateProcess*). Numai unul din aceste evenimente se poate petrece într-un spațiu de adresă, la un moment dat. Aceasta înseamnă aplicarea următoarelor restricții:

- Pe parcursul rutinelor de pornire a procesului și de inițializare a bibliotecilor DLL, pot fi create noi fire de execuție, dar ele nu încep execuția până când nu este realizată inițializarea bibliotecilor DLL.
- Numai un singur fir într-un proces poate fi prezent la un moment dat în inițializarea unui DLL sau a unei rutine detașate.
- Funcția *ExitProcess* nu se încheie până când nu mai există fire în inițializările de DLL sau în rutinele detașate.

1379 EXTINDEREA PRIN PROCESE COPIL C/C++

Așa cum ați învățat, programele dumneavoastră pot utiliza funcția *CreateProcess* pentru începerea executării unui nou proces. Când proiectați aplicații mai complexe, veți întâlni situații în care veți dori ca un alt bloc de cod să execute o operație. Programul dumneavoastră poate apela o anumită funcție pentru a efectua această operație. Însă funcțiile sunt executate în serie, deci codul nu poate continua executarea până ce o funcție nu își încheie execuția.

O metodă alternativă de executare a unui alt bloc de cod este crearea unui nou fir în cadrul procesului care preia execuția. Utilizarea firelor multiple, permite codului din programele dumneavoastră să continue prelucrarea în timp ce noul fir efectuează operația cerută. Din păcate, utilizarea firelor multiple va cauza probleme de sincronizare când un fir trebuie să utilizeze rezultatele noului fir. Veți învăța mai mult despre sincronizare în secțiunile viitoare.

O altă abordare este generarea de procese noi (denumite *procese copil*). Operarea cu procese permite programelor dumneavoastră să continue executarea operațiilor procesului copil pe o anumită problemă sau permite programelor să-și suspende execuția până ce procesul copil operează asupra unei anumite probleme. În următoarele două secțiuni, veți opera cu procese copil și cu fire de execuție. Lucrând cu ambele veți înțelege mai ușor diferențele dintre ele, precum și modul lor de utilizare în programele dumneavoastră.

Pe măsură ce programele Windows devin mai complexe, veți lucra tot mai mult cu componente partajate, independente unele de celelalte. Aceste componente pot fi de exemplu, biblioteci cu legare dinamică al căror cod este executat în procesul activ curent (obiecte in-process) sau servere automate OLE care se execută în afara programului (obiecte out-of-process). Cartea de față nu va discuta în detaliu despre obiectele in-process sau obiectele out-of-process.

ALTE OPERAȚII CU PROCESE COPIL

C/C++1380

Așa cum ați învățat, programele dumneavoastră pot controla modul de interacționare cu procesele copil. Însă, aproape toate procesele copil solicită accesul la datele cuprinse în spațiul de adresă al procesului părinte. În general, când un proces copil solicită acces la datele procesului părinte, trebuie să rulați procesul copil în propriul său spațiu de adresă și să oferiți acces la datele relevante din spațiul de adresă al procesului părinte. Controlarea accesului la spațiul de adresă al procesului părinte vă permite protejarea la coruperea accidentală a datelor nerelevante pentru procesul copil. Win32 vă oferă mai multe metode de transferare a datelor între procese: schimb dinamic de date (DDE), legare și încapsulare dinamică de obiecte (OLE), canale de transfer (pipes), sloturi de mesaje (Mailslots) etc. Una dintre cele mai convenabile (și mai simple) metode de partajare a datelor este *fișierul de mapare a memoriei*.

Un fișier de mapare a memoriei este un tip special de fișier care vă permite să rezervați o regiune de spațiu de adresă și să realizați stocarea fizică în acea regiune, asemănător alocării de memorie virtuală. Însă spre deosebire de memoria virtuală, stocarea fizică din fișierele de mapare a memoriei se realizează cu un fișier deja existent pe disc, și nu în fișierul de paginare al sistemului. După maparea fișierului, puteți accesa întregul fișier ca și cum programul l-ar fi încărcat în memorie.

Veți utiliza fișierele de mapare în următoarele trei scopuri:

- Sistemul utilizează fișiere de mapare a memoriei pentru încărcarea și executarea fișierelor cu biblioteci cu legare dinamică. Utilizarea fișierelor de mapare a memoriei conservă spațiu în fișierele de paginare și reduce timpul cerut de aplicație pentru începerea execuției.
- Puteți utiliza fișiere de mapare a memoriei pentru accesarea fișierelor de date pe disc. Utilizarea fișierului de mapare vă scutește de efectuarea operațiilor de I/O cu fișiere și stocarea conținutului lor în buffere.

- Puteți utiliza fișiere de mapare a memoriei pentru a permite ca procese multiple care rulează pe aceeași mașină, să partajeze date unele cu altele, așa cum deja ați aflat.

Multe obiecte de comunicație Win32 vor utiliza structura de fișier de mapare a memoriei pentru că e puternică și ușor de utilizat. Veți învăța mai mult despre crearea unui fișier de mapare a memoriei în secțiunile următoare.

Dacă doriți să creați un proces nou și să efectueze niște operații în timp ce programul părinte așteaptă rezultatul (de exemplu, scrie anumite date într-un fișier care va fi ulterior citit de procesul părinte), acest proces poate utiliza un cod similar cu următorul:

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

BOOL fSucces = CreateProcess(numeProces, &pi);
if (fSucces)
   {
     // inchide identificatorul de fir imediat ce nu mai aveti
     // nevoie de el
     CloseHandle(pi, hThread);
     WaitForSingleObject(pi, hProcess, INFINITE);
     // terminare proces
     GetExitCodeProcess(pi, hProcess, &dwExitCode);
     // inchide identificatorul de proces
     CloseHandle(pi, hProcess);
   }
```

Fragmentul de cod produce noul proces *numeProces*. Dacă reușește, fragmentul de cod va închide identificatorul de fir suplimentar pentru a elibera timpul CPU, apoi așteptă procesul să-și încheie executarea. După terminarea procesului, variabila *dwExitCode* conține informațiile de ieșire ale procesului, iar fragmentul de cod închide identificatorul procesului. Dacă fragmentul de cod nu reușește lansarea procesului, nu se va efectua nici o operație.

1381 RULAREA PROCESELOR COPIL DETAȘATE C/C++

În secțiunea 1380 ați învățat cum să creați un proces copil și să stopați execuția firului curent până când procesul se încheie. Totuși, de cele mai multe ori programele dumneavoastră vor porni procesele noi ca *procese copil detașate*. Un proces copil detașat este un proces creat de procesul părinte; după începerea executării procesului copil, procesul părinte fie nu mai necesită să intre în comunicare cu procesul copil, fie nu are nevoie ca procesul copil să se termine, înainte ca procesul părinte să continue operarea. Rularea proceselor copil detașate permit programelor dumneavoastră lansarea altor programe, fără nici un fel de preocupare în legătură cu timpul de administrare sau performanțele lor. Când executați un program din Windows Explorer de exemplu, exploratorul creează un nou proces. Apoi ignoră noul proces și își continuă propria prelucrare.

Când programul dumneavoastră creează un proces copil detașat, programul trebuie mai întâi să creeze procesul, apoi să închidă identificatorii noului proces și firul său primar. Următorul fragment de cod ilustrează modul în care programul dumneavoastră poate crea un nou proces copil detașat:

```
BOOL fSucces = CreateProcess(numeProces, &pi);
if (fSucces)
   {
   CloseHandle(pi, hThread);
   CloseHandle(pi, hProcess);
   }
```

FIRELE DE EXECUŢIE

C/C++1382

În secţiunile precedente, aţi învăţat cum să creaţi şi să gestionaţi procesele. Fiecare proces Win32 conţine unul sau mai multe fire de execuţie. Aţi reţinut că un fir este o cale de execuţie într-un proces. De fiecare dată când Windows iniţializează o nouă instanţă a unui proces, sistemul de operare creează un nou fir primar pentru acces proces. Firul primar porneşte o dată cu încărcarea programului în Windows. Firul la rândul său, va apela funcţia *WinMain* şi îşi va continua execuţia până când funcţia *WinMain* îşi încetează prelucrarea, iar programul apelează *ExitProcess* pentru închiderea sa. Pentru cele mai multe aplicaţii, firul primar creat de sistemul de operare este singurul fir cerut de aplicaţie. Cu toate acestea, pentru a-şi uşura operarea, procesele pot crea fire suplimentare. Ideea care stă la baza creării de fire suplimentare este folosirea la maxim şi cât mai eficient, a timpului de prelucrare al unităţii centrale. Când creaţi fire suplimentare, trimiteţi sistemului de operare cereri suplimentare pentru alocări de timp CPU. Aşa cum veţi învăţa în secţiunile următoare, firele suplimentare vor permite programelor dumneavoastră să efectueze mai eficient prelucrări de fundal, să efectueze calcule extinse şi activităţi bazate pe timp şi evenimente, precum şi alte operaţii de programare avansată. Secţiunile 1383 şi 1384 prezintă în detaliu când trebuie şi când nu trebuie să creaţi un fir de execuţie.

Când vă întrebaţi ce se execută în sistem la momentul curent, e util să consideraţi firele drept încapsulări în interiorul procesului. În funcţie de prioritatea firului, sistemul va prelucra unele fire mai des decât altele, atât în cadrul unui singur proces, cât şi în procese diferite. Figura 1382 prezintă un model logic al mai multor procese, fiecare conţinând mai multe fire care rulează în sistemul de operare Windows, ca şi ordinea potenţială în care CPU trebuie să prelucreze aceste fire.

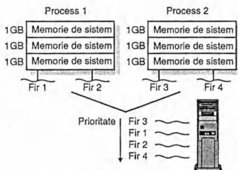


Figura 1382 *Modelul logic al firului de proces.*

1383 EVALUAREA NECESITĂȚII FIRELOR



Așa cum ați învățat în secțiunea 1382, programele dumneavoastră vor utiliza firele în primul rând pentru a se asigura că diverse secvențe ale unei aplicații vor beneficia de CPU cât mai mult posibil. Determinarea momentului când trebuie și când nu trebuie folosite fire suplimentare este una dintre cele mai importante decizii luate în timpul programării în Windows.

De exemplu, să considerăm un program de calcul tabelar. Un astfel de program trebuie să efectueze recalculări pe măsură ce utilizatorul introduce date în celule. Deoarece calculele pentru tabelele complexe pot dura mai multe secunde, o aplicație bine proiectată nu trebuie să recalculeze foaia de calcul după fiecare intrare de la utilizator. Aplicația trebuie să execute funcția de recalculare ca un fir separat, cu o prioritate inferioară față de cea a firului primar. Dacă utilizați două fire, firul primar va rula continuu în timp ce utilizatorul acționează tastele, astfel încât recalcularea de prioritate inferioară nu va avea acces la CPU. Când utilizatorul se oprește, se va executa firul cu prioritate inferioară, în timp ce firul primar așteaptă noile intrări de la utilizator.

Evident, puteți cel mai bine aplica utilitatea firelor la programele complexe care efectuează seturi de activități multiple. Activitățile de fundal fac uz din plin de firele suplimentare. Alte situații în care se face uz de fire sunt:

- Este util să creați un fir separat pentru gestionarea unei activități de tipărire în cadrul unei aplicații care, în timpul tipăririi, să permită utilizatorului să continue utilizarea aplicației.
- Puteți utiliza un fir separat pentru a întreține o casetă de dialog nemodală, care permite utilizatorului să întrerupă activități laborioase cum ar fi activitățile de copiere sau tipărire.
- Puteți utiliza fire pentru crearea de aplicații care simulează evenimente reale (de exemplu, evenimente care apar la un interval predeterminat de timp)

1384 CÂND NU TREBUIE CREAT UN FIR



Prima oară când mulți programatori obțin acces la un mediu cu suport multifir, ei au tendința să utilizeze prea mult firele, pur și simplu datorită noii forțe de prelucrare și funcționare pe care le pot exploata. Cei mai mulți programatori încep prin împărțirea aplicațiilor existente în fragmente mai mici, fiecare executând firul său. În ciuda faptului că acest procedeu de abordare a firelor poate părea util, el poate avea în realitate ca rezultat o risipă de timp de prelucrare. Fiecare fir reclamă realizarea unui anumit volum de operații și depunerea firului în stivă, chiar dacă sistemul de operare nu trimite firul la CPU pentru prelucrare. Firele suplimentare vor reduce performanțele sistemului deoarece, în plus față de operațiile firului, sistemul, în momentul alegerii firului pentru execuție, trebuie să testeze nivelul de prioritate al fiecărui fir activ curent.

Așa cum ați învățat, firele prezintă o nebănuită importanță și utilitate și își au locul lor în orice program Windows. Însă, este important să cunoaștem că, utilizând firele putem crea noi probleme, în timp ce le rezolvăm pe cele vechi. De exemplu, când concepeți un program de prelucrare a textelor și doriți ca funcția de imprimare să ruleze pe un fir propriu, prima operație este de a crea firul și de a imprima paginile documentului una câte una. Din nefericire, utilizatorul poate schimba documentul în timpul imprimării în fundal. În loc să

realizați soluția simplă la care va-ți fi așteptat, trebuie să copiați fișierul într-un fișier temporar, apoi să imprimați fișierul temporar, iar ulterior să-l ștergeți. Utilizarea firelor multiple împreună cu un fișier temporar este fără îndoială mai eficient pentru utilizator și face programul mai atractiv. În schimb, trebui să vă asigurați că realizarea firelor suplimentare nu dăunează procesului de prelucrare efectuat de program în firele sale curente.

Există o serie de reguli de care trebuie să țineți seama atunci când optați sau nu pentru crearea firelor multiple:

- Cu foarte rare excepții, toate componentele interfeței utilizator (controale și ferestre) trebuie să partajeze un fir comun.
- Programele trebuie să creeze fire în măsura în care are nevoie de ele și să evite crearea de fire suplimentare și „ținerea lor în rezervă".
- Programele trebuie să elibereze firele după terminarea procesului lor de prelucrare.
- Aplicațiile mai complexe care utilizează ferestre multiple, pot solicita fire suplimentare pentru administrarea prelucrărilor în cadrul anumitor ferestre.
- Nu trebuie create fire care permit utilizatorului să întrerupă un proces important al sistemului, care poate avea drept consecințe coruperea memoriei sau a altor prelucrări curente.

Ca regulă, trebuie întotdeauna să fiți sigur că aveți nevoie de un fir sau că firul va optimiza semnificativ prelucrarea programului. Utilizarea firelor suplimentare fără acest discernământ va avea ca efect pierderi în timpul de prelucrare și încetinirea programelor.

CREAREA UNUI FIR SIMPLU

C/C++1385

În secțiunile precedente ați învățat că programele dumneavoastră pot utiliza fire multiple care le permit să execute activități multiple în cadrul unui singur proces. Când creați fire de execuție, programele dumneavoastră vor utiliza funcția *CreateThread*. Funcția *CreateThread* creează un fir de execuție în spațiul de adresă al procesului apelant. Prototipul funcției *CreateThread* este următorul:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // pointer la
              // atributele de securitate ale firului
    DWORD dwStackSize, // dimensiunea stivei firului initial,
                   // in octeti
    LPTHREAD_START_ROUTINE lpStartAddress, // pointer catre
                                    // functia fir
    LPVOID lpParameter,     // argument pentru noul fir
    DWORD dwCreationFlags, // indicatoare de producere
    LPDWORD lpThreadId      // pointer catre identificatorul
                            // returnat de fir
);
```

Funcția *CreateThread* acceptă parametrii prezentați în Tabelul 1385.

Parametru	Descriere
lpThreadAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care determină dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpThreadAttributes* este NULL, identificatorul nu poate fi moștenit. Windows NT: membrul *lpSecurityDescriptor* din structură specifică descriptorul de securitate al noului fir. Dacă *lpThreadAttributes* este NULL, firul preia un descriptor implicit de securitate. Windows 95: membrul *lpSecurityDescriptor* din structură este ignorat.
dwStackSize	Specifică dimensiunea în octeți a stivei noului fir. Dacă este specificat 0, dimensiunea stivei preia ca dimensiune implicită aceiași dimensiune ca primul fir din proces. Stiva este alocată automat în spațiul de memorie al procesului și este eliberată când este terminat procesul. Rețineți că dimensiunea stivei crește, dacă este necesar. *CreateThread* încearcă să partajeze numărul de octeți specificat de *dwStackSize* și eșuează dacă dimensiunea depășește memoria disponibilă.
lpStartAddress	Începutul de adresă al noului fir. Aceasta este de regulă adresa funcției declarate cu convenția de apelare WINAPI care acceptă ca argument un singur pointer pe 32 de biți și returnează un cod de ieșire (exit) pe 32 de biți. Prototipul ei este *DWORD WINAPI ThreadFunc(LPVOID);*
lpParameter	Specifică o singură valoare de parametru pe 32 de biți transmisă firului.
dwCreationFlags	Specifică indicatoare suplimentare care controlează crearea firului. Dacă este specificat indicatorul *CREATE_SUSPENDED*, firul este creat în stare de suspendare și nu va rula până când nu este apelată funcția *ResumeThread*. Dacă această valoare este zero, firul rulează imediat după ce a fost creat. Windows nu acceptă nici o altă valoare la acest moment.
lpThreadId	Indică o variabilă pe 32 de biți care primește identificatorul firului.

Tabelul 1385 *Parametrii funcției* ***CreateThread***.

Dacă funcția *CreateThread* reușește, valoarea returnată este un identificator pentru noul fir. Dacă funcția eșuează, valoarea returnată va fi NULL. Sub Windows 95, funcția *CreateThread* reușește numai dacă este apelată în contextul unui program pe 32 de biți. O bibliotecă cu legare dinamică pe 32 de biți (DLL) nu poate crea un fir suplimentar dacă acel DLL este apelat de un program pe 16 biți.

Identificatorul noului fir este creat cu acces deplin la noul fir. Dacă apelul la *CreateThread* nu furnizează un descriptor de securitate, identificatorul returnat poate fi utilizat în orice funcție care necesită un obiect identificator de fir. Dacă descriptorul de securitate este furnizat de proces, înainte ca accesul să fie permis, se efectuează un test de acces la toate utilizările ulterioare ale identificatorului. Dacă testul de acces interzice accesul, procesul solicitant nu poate utiliza identificatorul pentru accesarea firului.

Execuția firului începe la funcția specificată de parametrul *lpStartAddress*. Dacă funcția se returnează, valoarea *DWORD* returnată este utilizată la terminarea firului într-un apel implicit la funcția *ExitThread*. Programul ar trebui să utilizeze funcția *GetExitCodeThread* pentru a obține valoarea returnată a firului după încheierea sa.

Funcția *CreateThread* poate reuși chiar dacă parametrul *lpStartAddress* indică date, cod sau nu este accesibil. Dacă adresa de început nu este validă când rulează firul, se semnalează o excepție, iar firul este terminat. Terminarea firului datorată unei adrese de început nevalide este gestionată ca eroare de ieșire (exit) pentru procesul acelui fir. Comportamentul este asemănător cu natura asincronă a funcției *CreateProcess*, unde procesul este creat chiar dacă face referire la o bibliotecă cu legare dinamică (DLL) absentă sau nevalidă.

Firul este creat cu prioritatea *THREAD_PRIORITY_NORMAL*. Utilizați *GetThreadPriority* și *SetThreadPriority* pentru a obține și fixa valoarea de prioritate pentru un fir.

Obiectul firului rămâne în sistem până când firul este terminat și toți indicatorii către el sunt închiși prin apelul la *CloseHandle*. Programul serializează *ExitProcess*, *ExitThread*, *CreateThread*, *CreateRemoteThread* împreună cu procesul care începe să ruleze (ca rezultat al apelului la *CreateProcess*). Numai unul din aceste evenimente se poate petrece într-un spațiu de adresă, la un moment dat. Aceasta înseamnă aplicarea următoarelor restricții:

- Pe parcursul rutinelor de pornire a procesului și de inițializare a bibliotecilor DLL, pot fi create noi fire de execuție, dar ele nu încep execuția până când nu sunt inițializate bibliotecile DLL.
- Numai un singur fir într-un proces poate fi prezent la un moment dat în inițializarea unui DLL sau a unei rutine detașate.
- Funcția *ExitProcess* nu se returnează până când nu mai există fire în inițializările de DLL sau în rutinele detașate.

Observație: *Un fir care utilizează funcții din biblioteci run-time de C ar trebui să folosească funcțiile de run-time* ***beginthread*** *și* ***endthread*** *și nu* ***CreateThread*** *și* ***ExitThread*** *pentru gestionarea firelor. A nu proceda astfel poate avea drept urmări mici scurgeri de memorie când se apelează* ***ExitThread***.

Pentru a înțelege mai bine prelucrarea efectuată de funcția *CreateThread*, să analizăm programul *Simple_Thread.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Simple_Thread* creează un nou fir de fiecare dată când utilizatorul selectează opțiunea *Test!* și afișează în fereastră că a început un nou fir. La selectarea opțiunii *Exit*, programul eliberează fiecare fir în parte și semnalează acest fapt. Următorul cod din *WndProc* creează firul:

```
Case IDM_TEST:          // porneste un fir
    {
        DWORD dwChildId;
        CreateThread(NULL, 0, ChildThreadProc, hWnd, 0, &dwChildId);
    }
    break;
```

VIZUALIZAREA LANSĂRII FIRULUI

C/C++ 1386

Așa cum ați învățat, programul dumneavoastră poate crea fire pentru efectuarea unor prelucrări suplimentare în propriul lor spațiu de execuție. De fiecare dată când creați un nou fir, Windows efectuează o serie de operații de bază pentru inițializarea firului și începerea prelucrării.

În primul rând, Windows alocă fiecărui fir propria lui stivă. Windows alocă stiva din propriul său spațiu adresă de 4Gb. Când programul utilizează variabile statice sau globale, firele multiple pot accesa aceste variabile simultan, putând corupe conținutul variabilelor. Windows creează, însă, în locul variabilelor globale, variabile locale și automatice în stiva firului, reducând considerabil eventualitatea coruperii lor. Așa cum ați învățat, trebuie întotdeauna să încercați utilizarea variabilelor locale și automatice în loc de variabile globale. Această regulă funcționează chiar mai bine în condițiile operării cu fire de execuție.

În al doilea rând, Windows alocă fiecărui fir propriul set de registre CPU, denumit *context* al firului. Windows stochează contextul firului într-o structură *CONTEXT*. Programul poate interoga structura *CONTEXT* în orice moment pentru determinarea stării registrelor CPU ale firului. Când sistemul de operare distribuie timp CPU pentru un fir, sistemul inițializează registrele CPU cu contextul firului. Registrele CPU conțin atât un pointer de instrucțiune (care identifică adresa următoarei instrucțiuni CPU de executare a firului), cât și un pointer de stivă (care identifică adresa stivei firului).

După ce firul încheie inițializările de stivă și context (presupunând că nu l-ați creat într-o stare de așteptare), firul va porni execuția cu prima linie a funcției definite la crearea firului.

1387 PAȘII EFECTUAȚI DE SISTEMUL DE OPERARE LA CREAREA FIRULUI



Așa cum ați învățat în secțiunea 1386, sistemul de operare efectuează o serie de pași importanți la alocarea firelor, din care programul va avea de câștigat. Însă, sistemul de operare efectuează în realitate șase pași specifici de fiecare dată când creează un nou fir, după cum urmează:

1. Alocă un obiect de nucleu al firului pentru identificarea și gestionarea firului nou creat. Obiectul nucleu deține multe informații ale sistemului pentru gestionarea firului. Identificatorul pentru obiectul nucleu al firului este valoarea returnată de *CreateThread*.
2. Inițializează codul de ieșire (exit) cu *STILL_ACTIVE* (cod menținut de Windows în obiectul nucleu al firului) și stabilește iterația de așteptare a firului la 1 (de asemenea, menținută de Windows în obiectul nucleu al firului).
3. Alocă structura *CONTEXT* pentru noul fir.
4. Pregătește stiva firului prin rezervarea unei regiuni de spațiu de adresă, cu angajarea a două pagini de stocare fizică în regiune, fixând valoarea de protecție a stocării la *PAGE_READWRITE* și stabilind atributul *PAGE_GUARD* la pagina a doua de sus.
5. Plasează valorile *lpStartAddr* și *lpvThread* la vârful stivei, astfel încât noul fir le vede ca parametri transmiși funcției *StartOfThread* (numai când codul utilizează biblioteca C run-time).
6. Inițializează registrul pointer de stivă în structura *CONTEXT* a firului pentru a indica valorile plasate de Windows în stivă (pasul 5). Apoi, sistemul de operare inițializează registrul pointer de instrucțiune pentru a indica funcția internă pe care Windows o prelucrează înainte de executarea primei instrucțiuni din funcția de pornire a firului.

1388 *CUM DETERMINĂM DIMENSIUNEA STIVEI DE FIR* C/C++

Așa cum ați învățat în secțiunea 1385, programele dumneavoastră pot specifica dimensiunea stivei firului în funcția *CreateThread*. Este important de reținut că, după ce Windows creează firul, programul dumneavoastră nu poate schimba în siguranță dimensiunea stivei firului. În schimb, Windows va crește în mod dinamic stiva când va fi necesar.

Dacă nu specificați o dimensiune pentru stiva firului, Windows va aloca stivei aceeași dimensiune cu a firului primar. *CreateThread* va crea stiva în spațiul de adresă de memorie al procesului. Puteți vizualiza crearea stivei într-un model logic al spațiului de stivă alocat, expus mai jos:

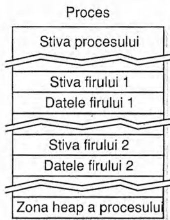


Figura 1388 *Spațiile de stivă pentru proces, firul primar și firul secundar.*

Când Windows alocă spațiu de stivă pentru fire suplimentare, o va face sub spațiul stivei de proces, la distanță de un segment. Windows va aloca în mod virtual spațiul de stivă a firului, astfel încât să poată deplasa stiva firului potrivit necesităților. De exemplu, dacă procesul inițializează firul secundar, apoi creează o matrice multidimensională *automatică* foarte cuprinzătoare (să zicem de [1024,64] de caractere), Windows trebuie să aibă capacitatea de a deplasa stiva secundară pentru a preveni ca procesul primar să suprascrie stiva secundară.

OBȚINEREA UNUI IDENTIFICATOR PENTRU FIRUL SAU PROCESUL CURENT C/C++ 1389

Pe măsură ce programele dumneavoastră devin mai complexe, se vor ivi situații când ele vor trebui să obțină în mod dinamic un identificator pentru firul sau procesul curent. Efectuarea oricăreia din aceste operații este relativ simplă. Programul poate apela oricând funcțiile *GetCurrentThread* sau *GetCurrentProcess* pentru obținerea unui pseudo-identificator al firului sau procesului curent (este numit pseudo-identificator pentru că valoarea sa are sens numai în firul sau procesul curent). Aceste funcții se implementează după prototipul descris mai jos:

```
HANDLE GetCurrentThread(VOID);
HANDLE GetCurrentProcess(VOID);
```

Trebuie să rețineți că pseudo-identificatorii returnați de ambele funcții nu au nici un fel de utilitate în afara procesului curent. Pentru a transmite identificatorul unui fir sau altui proces, programul trebuie să utilizeze funcția *DuplicateHandle*.

Pentru a înțelege mai bine prelucrările efectuate de funcțiile *GetCurrentThread* și *GetCurrentProcess* să analizăm programul *Show_Current.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Show_Current* creează unul câte unul firele și afișează informații despre firele sau procesele respective.

1390 GESTIONAREA TIMPULUI DE PRELUCRARE A FIRULUI

C/C++

Așa cum probabil vă imaginați, determinarea duratei de timp de prelucrare a unei activități date într-un mediu multifir este semnificativ mai dificilă decât într-un mediu cu fir unic. Datorită faptului că procesul poate menține un fir ocupat pentru recalcularea unui algoritm complex, în timp ce firele în celelalte procese continuă să-și dispute timpul din CPU, procesul în cauză poate consuma un timp mai mare de executare, între două situații de referință. Programele trebuie să utilizeze o metodă diferită pentru înregistrarea timpului de prelucrare a unui fir, față de codul simplu de cronometrare a execuției programului, așa cum ați procedat în secțiunea 625, în programul *clock.c*.

Programele trebuie să utilizeze o funcție care testează durata de execuție al firului. În Windows, funcția care execută această prelucrare este *GetThreadTimes*. Funcția *GetThreadTimes* obține informații de durată a unui fir specificat. Veți utiliza funcția după prototipul prezentat mai jos:

```
BOOL GetThreadTimes(
    HANDLE hThread, // specifica firul de interes
    LPFILETIME lpCreationTime, // cand a fost creat firul
    LPFILETIME lpExitTime, // cand a fost distrus firul
    LPFILETIME lpKernelTime, // timpul petrecut in modul kernel
                             // (nucleu)
    LPFILETIME lpUserTime    // timpul petrecut in modul user
                             // (utilizator)
);
```

Funcția *GetThreadTimes* acceptă parametrii prezentați în tabelul 1390.

Parametru	Descriere
hThread	Un identificator deschis care specifică firul pentru care se caută informația referitoare la durată. Acest identificator trebuie creat cu accesul *THREAD_QUERY_INFORMATION*.
lpCreationTime	Indică o structură *FILETIME* care primește momentul de creare a firului.
lpExitTime	Indică o structură *FILETIME* care primește momentul de ieșire a firului. Dacă firul nu a ieșit, conținutul structurii este nedefinit.

Parametru	**Descriere**
lpKernelTime	Indică o structură *FILETIME* care primește durata de execuție a firului în modul kernel.
lpUserTime	Indică o structură *FILETIME* care primește durata de execuție a firului în modul user.

Tabelul 1390 *Parametrii funcției* ***GetThreadTimes.***

Dacă funcția reușește, valoarea returnată este diferită de zero. Dacă funcția eșuează, valoarea returnată este zero. Funcția *GetThreadTimes* folosește structurile de date *FILETIME* pentru exprimarea tuturor timpilor. Astfel de structuri conțin două valori de 32 biți care se combină pentru a forma un contor de 64 de biți de unități de timp de 100 de nanosecunde. Momentele de creare și de încheiere ai firului sunt momente de timp exprimate în structura *FILETIME* ca durată scursă de la miezul nopții de 1 ianuarie 1601, la meridianul Greenwich, Anglia. Interfața Win32 API oferă o serie de funcții pe care aplicațiile le pot utiliza pentru convertirea acestor valori la forme de utilitate mai generală.

Duratele din modurile kernel și utilizator sunt intervale de timp exprimate în nanosecunde. De exemplu, dacă un fir a stat o secundă în modul kernel, funcția *GetThreadTimes* va completa structura *FILETIME* specificată de parametrul *lpKernelTime* cu o valoare de zece milioane pe 64 de biți, ceea ce corespunde numărului de unități de 100 de nanosecunde într-o secundă.

GESTIONAREA TIMPULUI DE PRELUCRARE AL FIRELOR MULTIPLE

C/C++1391

După cum ați învățat în secțiunea 1390, programele pot utiliza funcția *GetThreadTimes* pentru a determina durata de execuție a firului. Deseori, însă, programele vor necesita informații detaliate despre durata de execuție a mai multor fire din același proces (pentru a determina dacă toate firele din proces sunt lente sau dacă unul sau mai multe fire reprezintă cauza unei încetiniri a procesului). În aceste situații, programele pot utiliza funcția *GetProcessTimes* pentru a obține informații cu timpii unui proces specificat.

Prototipul funcției GetProcessTimes este:

```
BOOL GetProcessTimes(
    HANDLE hProcess, // specifica procesul de interes
    LPFILETIME lpCreationTime, // cand a fost creat procesul
    LPFILETIME lpExitTime, // cand s-a incheiat procesul
    LPFILETIME lpKernelTime, // durata in modul kernel (nucleu)
    LPFILETIME lpUserTime // durata in modul user (utilizator)
);
```

Funcția *GetProcessTimes* acceptă parametrii prezentați în tabelul 1390.

Dacă funcția reușește, ea returnează o valoare diferită de zero. Dacă funcția eșuează, ea returnează valoarea zero. Funcția *GetProcessTimes* folosește structurile de date *FILETIME* pentru a exprima toate duratele și momentele. Asemenea structuri conțin două valori pe 32 de biți care formează un contor pe 64 de biți de unități de timp de 100 de nanosecunde. Momentele de creare și de încheiere ale procesului sunt exprimate de structuri *FILETIME* ca durate scurse de la miezul nopții din 1 ianuarie 1601, meridianul Greenwich, Anglia. Win32 API oferă mai multe funcții ce pot fi folosite de o aplicație pentru a converti astfel de valori în formate mai utile.

Duratele din modul kernel şi user ale procesului sunt intervale de timp. De exemplu, dacă un proces a stat o secundă în modul kernel, funcţia *GetProcessTimes* va pune în structura *FILETIME* specificată de parametrul *lpKernelTime* cu o valoare pe 64 de biţi de 10 milioane, numărul de unităţi de 100 de nanosecunde dintr-o secundă.

CD-ROM-ul care însoţeşte cartea conţine programul *Show_ThPr_Times*, care deschide o serie de fire şi returnează durata de procesare a fiecărui fir, precum şi întreaga durată de prelucrare a programului.

1392 FUNCŢIA GETQUEUESTATUS



Când lucraţi cu fire, veţi întâmpina situaţii în care apar evenimente (cum ar fi acţiuni de la tastatură şi altele) pe care firul primar le va prelucra de regulă în funcţia de prelucrare a mesajului din *WndProc*. Dacă însă aveţi un fir copil suspendat, controlul informaţiilor din mesaj devine mai complicat. Însă, deoarece firul este deja suspendat (din diferite motive), funcţia de prelucrare a mesajului firului va reţine aceste mesaje în coada de mesaje a firului respectiv. Pentru determinarea conţinutului cozii de mesaje, după ce un fir suspendat reia execuţia, programele pot utiliza funcţia *GetQueueStatus*. Această funcţie returnează indicatoare care indică tipurile de mesaje regăsite în coada de mesaje a firului apelant. Implementarea funcţiei *GetQueueStatus* se face potrivit prototipului prezentat în continuare:

```
DWORD GetQueueStatus(UINT flags    // indicatoare);
```

Parametrul *flags* specifică indicatoarele de stare ale cozii de mesaje care dau tipurile de mesaje pe care funcţia le poate testa Acest parametru poate fi o combinaţie a valorilor descrise în Tabelul 1392.

Valoare	**Semnificaţie**
QS_ALLEVENTS	În coadă pot fi următoarele evenimente : o intrare, *WM_TIMER*, *WM_PAINT*, *WM_HOTKEY* sau un mesaj expediat.
QS_ALLINPUT	În coadă este un mesaj, de orice tip.
QS_HOTKEY	Un mesaj *WM_HOTKEY* este în coadă.
QS_INPUT	În coadă se află un mesaj de intrare.
QS_KEY	În coadă se află mesaje *WM_KEYUP*, *WM_KEYDOWN*, *WM_SYSKEYUP* sau *WM_SYSKEYDOWN*.
QS_MOUSE	În coadă se află mesajul *WM_MOUSEMOVE* sau mesaje de la un buton de mouse (*WM_LBUTTONUP*, *WM_RBUTTONDOWN* etc.).
QS_MOUSEBUTTON	Un mesaj de la un buton de mouse (*WM_LBUTTONUP*, *WM_RBUTTONDOWN* etc.).
QS_MOUSEMOVE	În coadă se află mesajul *WM_MOUSEMOVE*.
QS_PAINT	În coadă se află mesajul *WM_PAINT*.
QS_POSTMESSAGE	În coadă se află un mesaj expediat (altele decât cele deja menţionate).
QS_SENDMESSAGE	În coadă se află un mesaj trimis de un alt fir sau aplicaţie.
QS_TIMER	În coadă se află mesajul *WM_TIMER*.

Tabelul 1392 *Valorile acceptate pentru parametrul flags.*

Cuvântul mai semnificativ al valorii returnate indică tipurile de mesaje aflate la acel moment în coadă. Cuvântul mai puţin semnificativ indică tipurile de mesaje care au fost adăugate în coadă de Windows şi care se află încă în coadă de la ultimul apel la funcţiile *GetQueueStatus*, *GetMessage* sau *PeekMessage*.

Prezenţa indicatorului *QS_* în valoarea returnată nu garantează că un apel succesiv la funcţiile *PeekMessage* sau *GetMessage* va returna un mesaj. *GetMessage* şi *PeekMessage* efectuează o filtrare internă care poate avea ca efect prelucrarea internă a mesajului. Pentru acest motiv, valoarea returnată de funcţia *GetQueueStatus* trebuie considerată numai ca o indicaţie a necesităţii apelării funcţiilor *GetMessage* sau *PeekMessage*.

Pentru a înţelege mai bine prelucrarea efectuată de funcţia *GetQueueStatus* să analizăm programul *Queue_Status.cpp* conţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *Queue_Status* creează un fir, apoi întrerupe execuţia cinci secunde, astfel încât utilizatorul să genereze evenimente care sunt expediate de Windows în coada de mesaje. După reluarea firului, programul afişează evenimentele din coadă.

GESTIONAREA EXCEPŢIILOR NETRATATE C/C++1393

Sistemul de operare Win32 plasează un handler primar pentru excepţii în fruntea fiecărui fir sau proces. Scopul unui handler primar de excepţii este să asigure că programul răspunde convenabil la excepţii netratate (de exemplu, programul s-a închis fără un impact negativ asupra celorlalte procese). Uneori, va fi nevoie să captaţi toate excepţiile netratate într-o rutină specială, posibil o rutină care salvează lucrarea unui utilizator pe disc, într-un fişier de recuperare. Funcţia *SetUnhandledExceptionFilter* permite aplicaţiei să înlocuiască programul handler primar de excepţii pe care Win32 îl plasează în fruntea fiecărui fir sau proces.

După apelarea funcţiei *SetUnhandledExceptionFilter*, dacă apare o excepţie într-un proces pe care Windows nu îl depanează în acel moment, iar excepţia se face în filtrul de excepţii netratate din Win32, filtrul respectiv va apela funcţia de filtrare a excepţiilor specificată în parametrul *lpTopLevelExceptionFilter*. Utilizarea funcţiei *SetUnhandledExceptionFilter* se va face potrivit prototipului prezentat mai jos.

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
   LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter );
```

Parametrul *lpTopLevelExceptionFilter* furnizează adresa unei funcţii primare de filtrare a excepţiilor care va fi apelată de fiecare dată când funcţia *UnhandledExceptionFilter* obţine controlul, iar procesul nu este în curs de depanare. Valoarea NULL a acestui parametru specifică gestionarea implicită în *UnhandledExceptionFilter*.

Funcţia de filtrare are o sintaxă similară cu funcţia *UnhandledExceptionFilter*. Funcţia deţine un singur parametru de tip *LPEXCEPTION_POINTERS* şi returnează o valoare de tip *LONG*. Funcţia de filtrare returnează una din valorile descrise în Tabelul 1393.

Valoare	Semnificație
EXCEPTION_EXECUTE_HANDLER	Returnează controlul din *UnhandledExceptionFilter* și execută programul handler de excepții asociat. De regulă, rezultatul este terminarea procesului.
EXCEPTION_CONTINUE_EXECUTION	Returnează controlul din *UnhandledExceptionFilter* și continuă executarea de la punctul excepției. Rețineți că funcția de filtrare este liberă să modifice starea de continuitate prin modificarea informațiilor referitoare la excepții prin parametrul *LPEXCEPTION_POINTERS*.
EXCEPTION_CONTINUE_SEARCH	Continuă cu executarea normală a funcției *UnhandledExceptionFilter*. Aceasta înseamnă conformarea la indicatoarele *SetErrorMod* sau invocarea casetei pop-up de mesaje *Application Error*.

Tabelul 1393 *Valorile returnate posibile ale funcției de filtrare.*

Funcția *SetUnhandledExceptionFilter* returnează adresa filtrului de excepții anterior stabilit de funcție. Valoarea returnată NULL semnifică faptul că nu există un handler primar de excepții. Execuția lui *SetUnhandledExceptionFilter* înlocuiește filtrul primar de excepții pentru toate firele viitoare ale procesului apelant. Programul handler de excepții specificat de *lpTopLevelExceptionFilter* este executat de firul care a provocat eroarea. Deoarece programul handler de excepții se execută în cadrul firului, acesta poate afecta capacitatea programului handler de excepții de recuperare din anumite excepții, cum ar fi o stivă nevalidă.

1394 TERMINAREA FIRELOR



Pe măsură ce programele dumneavoastră devin mai complexe, se vor ivi situații când un fir nu-și va încheia execuția în mod normal. Așa cum ați învățat, programele trebuie să oprească sau să întrerupă un fir pentru a opri sistemul de operare de la programarea firelor pentru execuție. Dacă programele dumneavoastră nu își încheie execuția corect, ele nu se vor închide. În schimb, programele trebuie să ceară sistemului de operare să termine firele. Pentru aceasta se utilizează funcția *TerminateThread*. Funcția se implementează potrivit următorului prototip:

```
BOOL TerminateThread(
    HANDLE hThread, // identificator pentru fir
    DWORD dwExitCode // cod de iesire pentru fir
);
```

Parametrul *hThread* identifică firul care trebuie terminat. Sub Windows NT, identificatorul trebuie să dețină accesul *THREAD_TERMINATE*. Parametrul *dwExitCode* conține codul de ieșire pentru fir. Utilizați funcția *GetExitCodeThread* pentru a prelua valoarea de ieșire a firului. Dacă funcția reușește, valoarea returnată este diferită de zero. Dacă funcția eșuează, valoarea returnată este zero.

Programul dumneavoastră utilizează *TerminateThread* pentru a determina terminarea unui fir. Când aceasta se întâmplă, firul țintă nu are posibilitatea de a executa nici un cod în modul utilizator, iar stiva sa inițială nu este dezalocată. Bibliotecile DLL atașate firului nu sunt notificate că firul este în curs de terminare.

Funcția *TerminateThread* este o funcție potențial periculoasă. Ea trebuie utilizată în cele mai extreme situații. Trebuie să apelați funcția *TerminateThread* numai când cunoașteți exact ce face firul țintă și controlați întregul cod pe care firul țintă probabil îl rulează în timpul terminării. De exemplu, *TerminateThread* poate genera următoarele probleme:

- Dacă firul țintă posedă o secțiune critică, Windows nu o va elibera.
- Dacă firul țintă execută la terminare anumite apeluri ce țin de kernel32, starea acestui kernel32 în prelucrarea firului poate fi inconsistentă.
- Dacă firul țintă manipulează starea globală a unui DLL partajat, starea acestui DLL poate fi distrusă de Windows, afectând ceilalți utilizatori ai bibliotecii DLL.

Un fir nu se poate autoproteja împotriva funcției *TerminateThread*, decât prin controlarea accesului la identificatorii săi. Identificatorul firului returnat de funcțiile *CreateThread* și *CreateProcess* are acces *THREAD_TERMINATE*, astfel încât orice apelant care deține unul din acești identificatori poate termina firul. Dacă firul țintă este ultimul fir al procesului, la apelul funcției *TerminateThread*, procesul acelui fir este de asemenea terminat. Obiectul firului intră în starea de semnalizare, eliberând orice alte fire care erau în așteptarea terminării firului. Starea de terminare a firului se schimbă din valoarea *STILL_ACTIVE* în valoarea parametrului *dwExitCode*.

Terminarea unui fir nu elimină în mod necesar obiectul firului din sistem. Un obiect fir este șters când ultimul identificator al firului este închis.

Pentru a înțelege mai bine prelucrarea efectuată de funcția *TerminateThread*, analizați programul *Manip_Threads.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Manip_Threads* permite utilizatorului să creeze un fir și apoi să suspende acel fir (cu ALT+S), să reia acel fir (cu ALT+R) sau să termine firul (cu ALT+K). Programul evită problemele referitoare la terminarea firelor deoarece firul nu desfășoară nici o activitate ci doar permite utilizatorului accesul la cel mai recent fir creat. Funcțiile *ThreadProc* și *WndProc* ale programului *Manip_Threads* conțin codul care efectuează prelucrarea operațiilor.

DETERMINAREA IDENTIFICATORULUI ID AL UNUI FIR SAU PROCES

C/C++1395

Așa cum ați învățat în secțiunea 1389, programele dumneavoastră vor avea nevoie deseori de identificatori temporari, sau pseudo-handle, pentru firul sau procesul curent. Mai puțin frecvent, programele dumneavoastră vor avea nevoie de un identificator permanent sau de o altă valoare unică pentru reprezentarea unui fir sau proces în întregul sistem. Interfața Win32 API furnizează două funcții: *GetCurrentThreadId* și *GetCurrentProcessId*, care permit programelor să obțină o valoare unică *DWORD* utilizată de sistemul de operare pentru reprezentarea internă a firelor și proceselor. Aceste funcții se vor implementa după prototipurile prezentate mai jos:

```
DWORD GetCurrentThreadId(VOID);
DWORD GetCurrentProcessId(VOID);
```

Funcția *GetCurrentThreadId* returnează identificatorul de fir al firului apelant, care este valoarea returnată de firul apelant. Până la terminarea firului, identificatorul de fir identifică în mod unic firul în sistem.

Similar, funcția *GetCurrentProcessId* returnează identificatorul de proces al procesului apelant. Funcția nu are parametri. Valoarea returnată este identificatorul de proces al procesului apelant. Până la terminarea firului, identificatorul de proces identifică în mod unic procesul în sistem.

Programul *ShowCurrent* de la secțiunea 1389 utilizează ambele funcții *GetCurrentThreadId* și *GetCurrentProcessId*.

Observație: *Similar cu un identificator (handle), valoarea DWORD returnată de GetCurrentThreadID și GetCurrentProcessID este o valoare unică ce identifică firul sau procesul în tot sistemul de operare. Nu confundați identificatorul ID al procesului sau identificatorul ID al firului cu pseudo-identificatorii returnați de GetCurrentThread și GetCurrentProcess.*

1396 *MODUL CUM SISTEMUL DE OPERARE PROGRAMEAZĂ FIRELE*



Așa cum ați învățat, sistemul de operare Win32 operează în regim multi-fir. Sistemul de operare Win32 poate gestiona un mare număr de fire sau procese în succesiune strânsă. Însă, așa cum s-a indicat în secțiunile precedente, Win32 va gestiona unele fire mai rapid sau într-o ordine diferită față de altele. De exemplu, sistemul de operare Win32 va avea tendința de a atașa o prioritate mai înaltă firelor din procesul curent decât din procesele care se execută pe fundal.

De fapt, sistemul de operare Win32 programează fiecare fir care solicită prelucrări în CPU (adică toate firele active), pe baza nivelului lor de prioritate. Secțiunea 1397 va explica în detaliu nivelurile de prioritate. Când sistemul alocă unitatea centrală unui fir, el tratează toate firele de același nivel ca egale. Cu alte cuvinte, sistemul atribuie primului fir din coadă nivelul de prioritate 31 la un CPU, iar după ce secvența de timp a acestui fir expiră, sistemul atribuie următorului fir prioritatea 31 la un CPU. În consecință, e important de reținut că în cazul în care aveți întotdeauna cel puțin un fir cu prioritatea 31 pentru fiecare CPU, firele cu prioritate inferioară nu se vor executa niciodată. Programatorii denumesc această condiție de prioritate *saturație*. Saturația apare când unele fire utilizează atât de mult timp de prelucrare CPU, încât nici un alt fir nu se poate executa.

Când sistemul încheie firele cu prioritatea 31, va începe să atribuie firelor prioritatea 30. Când sistemul va termina firele cu prioritatea 30, va începe să atribuie firelor prioritatea 29 și așa mai departe. Se poate vedea atunci că firele cu prioritate mică nu vor fi, de regulă niciodată executate într-un astfel de sistem. Așa cum reiese, însă, deseori chiar și firele cu prioritatea cea mai înaltă nu solicită timp CPU, ceea ce eliberează unitatea centrală pentru gestionarea firelor cu prioritate mai mică.

În final, trebuie să înțelegeți că, în cazul în care este executat un fir cu prioritate mică – chiar dacă este la mijlocul secvenței sale de timp –, iar sistemul de operare determină că un fir de prioritate mai înaltă este în așteptarea execuției, sistemul va opri imediat execuția firului cu prioritate mai mică și va începe execuția firului cu prioritate mai înaltă. Firul cu prioritate înaltă va avea întotdeauna întâietate față de firele de prioritate mică, indiferent de ce operează firul de prioritate mică sau în ce stare se află la momentul execuției.

Observație: *Microsoft își rezervă dreptul de a schimba algoritmul pe care sistemul de operare bazat pe Win32 îl utilizează în planificarea firelor. Sistemele Windows 95, Windows*

NT 3.50 și Windows NT 4.0 utilizează fiecare un algoritm Win32 de planificare a firelor diferit. Dacă înțelegeți modul de operare cu priorități al firelor și îl veți utiliza cu atenție, programele dumneavoastră nu vor fi atât de legate de metoda fiecărui sistem de operare de a gestiona prioritatea firelor și care ar putea avea ca efect probleme serioase în eventualitatea în care Microsoft va schimba în viitoarele versiuni algoritmul de planificare.

NIVELURILE DE PRIORITATE

C/C++ 1397

Așa cum ați învățat în secțiunea 1396, sistemul de operare Win32 planifică toate firele active pe baza nivelului curent de prioritate. Nivelurile de prioritate se extind de la valoarea 0 (cel mai scăzut nivel de prioritate) la valoarea 30 (cel mai înalt nivel de prioritate). Sistemul de operare atribuie nivelul zero de prioritate unui fir special al sistemului cunoscut cu numele de *fir de pagină zero*. Firul de pagină zero este răspunzător de atribuirea valorii zero fiecărei pagini libere din sistem, atunci când nu mai există alte fire care trebuie executate în sistem. Nu este posibil pentru orice fir de a avea nivelul de prioritate zero, în afara firului de pagină zero.

Când creați fire, nu utilizați numere pentru a atribui niveluri de prioritate. În schimb, sistemul utilizează un proces în doi pași pentru determinarea nivelului de prioritate a firului. Primul pas este atribuirea unei *clase de prioritate* unui proces. Clasa de prioritate a procesului comunică sistemului prioritatea cerută de proces, comparativ cu alte procese în rulare. Secțiunea 1398 prezintă clasele de prioritate în detaliu. Al doilea pas este atribuirea unui nivel de prioritate fiecărui fir deținut de proces.

Când cerați prima dată un fir în cadrul procesului, nivelul de prioritate al firului este similar cu cel al procesului. În secțiunea 1400, veți învăța cum să utilizați interfața Win API pentru schimbarea nivelului de prioritate al unui fir.

CLASELE DE PRIORITATE WINDOWS

C/C++ 1398

Așa cum se explică în secțiunea 1397, Windows utilizează un proces în doi pași pentru determinarea priorității firelor. Primul pas este determinarea clasei de prioritate a procesului. Win32 acceptă patru clase de prioritate: inactivă (*idle-priority*), normală (*normal-priority*), înaltă (*high-priority*) și în timp real (*realtime-priority*). Tabelul 1398 detaliază clasele de prioritate:

Prioritate	Semnificație
HIGH_PRIORITY_CLASS	Indică un proces care efectuează activități urgente ce solicită o executare imediată pentru a rula corect. Firul de execuție al unui proces cu o clasă de prioritate *high-priority* are întâietate față de procesele cu clase de prioritate *normal-priority* sau *idle-priority*. Un exemplu este *Windows Task List* care trebuie să răspundă rapid când este apelată de utilizator, indiferent de încărcarea sistemu- lui de operare. Fiți extrem de atenți când utilizați clasa de prioritate *high-priority*, pentru că o aplicație cu această clasă de prioritate poate utiliza aproape toate ciclurile disponibile de CPU. Nivelul de prioritate al unui proces cu prioritatea *HIGH_PRIORITY_CLASS* este 13.

(continuare)

Prioritate	Semnificație
IDLE_PRIORITY_CLASS	Indică un proces ale cărui fire rulează numai când sistemul este inactiv și dau întâietate firelor oricărui proces care rulează într-o clasă de prioritate superioară. Un exemplu este un program de salvare pentru ecran. Clasa de prioritate *idle-priority* este moștenită de procesele copil. Nivelul de prioritate al unui proces cu prioritatea *IDLE_PRIORITY_CLASS* este 6.
NORMAL_PRIORITY_CLASS	Indică un proces normal, fără nici o cerință specială de programare a activității. Nivelul de prioritate al unui proces cu prioritatea NORMAL_PRIORITY_CLASS este 8.
REALTIME_PRIORITY_CLASS	Indică un proces care prezintă cea mai înaltă prioritate posibilă. Firele din clasa de prioritate *realtime-priority* au întâietate față de toate celelalte procese, inclusiv față de procesele sistemului de operare care efectuează operații importante. De exemplu, un proces în timp real care rămâne în execuție mai mult decât un foarte scurt interval de timp, poate avea ca efect imposibilitatea golirii rezervelor cache pe disc sau absența răspunsurilor de la mouse. Nivelul de prioritate al unui proces cu prioritatea *REALTIME_PRIORITY_CLASS* este 24.

Tabelul 1398 *Clasele de prioritate în* ***Win32***.

Este util să înțelegeți pe deplin impactul nivelurilor diferite de clase de prioritate. De exemplu, veți utiliza clasa *HIGH_PRIORITY_CLASS* numai în situațiile de absolută necesitate. Cel mai obișnuit proces care utilizează valoarea *HIGH_PRIORITY_CLASS* este *Windows Explorer*. Chiar dacă majoritatea firelor pentru desktop sunt inactive pe durata executării normale, utilizatorii se așteaptă ca solicitările de pe desktop să capete răspuns la manevrele de acces. Deci Windows conferă firelor din Explorer prioritate maximă și vor avea întâietate aproape asupra tuturor celorlalte fire atunci când utilizatorul selectează o opțiune din desktop. Când creați programe care utilizează valoarea *HIGH_PRIORITY_CLASS*, cererile din desktop nu vor avea răspuns, iar Windows chiar îl va bloca.

În general, programele dumneavoastră vor utiliza *IDLE_PRIORITY_CLASS* pentru aplicațiile de monitorizare a sistemului. De exemplu, puteți scrie o aplicație care va afișa periodic cantitatea de memorie RAM existentă în sistem. Deoarece nu doriți ca aplicația să interfereze cu performanțele altor activități mai importante, veți stabili pentru procesul periodic clasa *IDLE_PRIORITY_CLASS*.

Windows atribuie în mod automat valoarea *NORMAL_PRIORITY_CLASS* oricărui proces căruia nu îi atribuiți în mod explicit o altă valoare. În general, programele dumneavoastră trebuie să utilizeze valoarea *NORMAL_PRIORITY_CLASS*. Rețineți că, atunci când utilizatorul trimite procesul în fundal, sistemul de operare crește prioritatea relativă pentru realizarea unei viteze superioare de execuție. De exemplu, în Windows 95 sistemul de operare adaugă o unitate la numărul de prioritate al proceselor din fundal.

În general, programele dumneavoastră nu trebuie niciodată să utilizeze valoarea *REALTIME_PRIORITY_CLASS* deoarece prioritatea în timp real este o prioritate foarte înaltă, de fapt este chiar mai înaltă decât majoritatea firelor de gestionare a sistemului de operare. Firele din sistem care controlează mouse-ul și tastatura, salvarea pe disc în fundal, chiar și acționarea CTRL+ALT+DEL,

toate se execută cu o prioritate mai mică decât cea în timp real. Programele care utilizează prioritatea în timp real vor avea efecte adverse semnificative în sistemul utilizatorului.

MODIFICAREA CLASEI DE PRIORITATE A UNUI PROCES

C/C++1399

Așa cum ați învățat în secțiunea 1398, sistemul de operare Win32 atribuie automat prioritate normală fiecărui proces nou. Deseori însă, programele pot solicita schimbări în clasa de prioritate a procesului. Puteți utiliza funcțiile *GetPriorityClass* și *SetPriorityClass* pentru gestionarea claselor de prioritate. Funcția *GetPriorityClass* returnează clasa de prioritate pentru procesul specificat. Prototipul funcției este:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Parametrul *hProcess* identifică procesul. Sub Windows NT, identificatorul trebuie să aibă dreptul de acces *PROCESS_QUERY_INFORMATION*. Dacă funcția reușește, valoarea returnată este clasa de prioritate a procesului specificat. Dacă funcția eșuează, valoarea returnată este zero. Valoarea de prioritate a procesului este una dintre valorile prezentate în Tabelul 1398.

În mod asemănător, funcția *SetPriorityClass* stabilește clasa de prioritate a procesului specificat. Această valoare, împreună cu valoarea de prioritate a fiecărui fir din proces, determină nivelul de bază al priorității fiecărui fir. Prototipul funcției este:

```
BOOL SetPriorityClass(
    HANDLE hProcess,        // identificator pentru proces
    DWORD dwPriorityClass // valoarea clasei de prioritate
);
```

La fel ca funcția *GetPriorityClass*, identificatorul *hProcess* identifică procesul. Sub Windows NT, *hProcess* trebuie să aibă dreptul de acces *PROCESS_SET_INFORMATION*. Parametrul *dwPriorityClass* specifică clasa de prioritate a procesului și poate lua oricare din valorile prezentate în Tabelul 1398. Dacă funcția reușește, valoarea returnată este diferită de zero. În caz de eșec, valoarea returnată este zero.

Fiecare fir are un nivel de prioritate de bază pe care Windows îl determină pe baza valorii de prioritate a firului și a clasei de prioritate a procesului său. Sistemul utilizează nivelul de prioritate de bază al tuturor firelor executabile pentru a determina ce fir va primi următoarea secvență de timp CPU. Funcția *SetThreadPriority* vă permite să stabiliți nivelul de prioritate de bază a unui fir în raport cu clasa de prioritate a procesului său. Secțiunea 1400 utilizează funcția *SetThreadPriority* pentru a stabili nivelul de prioritate al unui fir.

CD-ROM-ul care însoțește cartea de față conține programul *Get_Set_Priority* care permite utilizatorului selectarea unei clase de prioritate a unui proces. După fiecare selecție, procesul va executa o funcție intens consumatoare de timp CPU, apoi afișează rezultatele împreună cu clasa de prioritate returnată de sistemul de operare.

1400 STABILIREA UNEI PRIORITĂȚI RELATIVE PENTRU UN FIR



Așa cum ați învățat, Windows stabilește un nivel de prioritate pe baza clasei de prioritate a procesului care deține firul și a nivelului de prioritate a firului. În secțiunea 1399, ați învățat cum să utilizați funcția *SetProcessClass* pentru schimbarea clasei de prioritate a procesului. Pentru schimbarea nivelului de prioritate a firelor unui proces, programul trebuie să utilizeze funcția *SetThreadPriority*. Funcția *SetThreadPriority* stabilește valoarea de prioritate pentru firul specificat. Această valoare, împreună cu clasa de prioritate a procesului de care aparține firul, determină nivelul de prioritate de bază al firului. Funcția *SetThreadPriority* se implementează conform prototipului prezentat mai jos:

```
BOOL SetThreadPriority(
    HANDLE hThread, // identificator pentru fir
    int nPriority // nivelul de prioritate al firului
);
```

Parametrul *hThread* identifică firul a cărui valoare de prioritate urmează a fi stabilită. Sub Windows NT, identificatorul trebuie să dețină dreptul de acces *THREAD_SET_INFORMATION*. Parametrul *nPriority* specifică valoarea de prioritate pentru fir. Acest parametru poate lua una din valorile prezentate în Tabelul 1400.

Prioritate	Semnificație
THREAD_PRIORITY_ABOVE_NORMAL	Indică 1 punct deasupra priorității normale a clasei de prioritate.
THREAD_PRIORITY_BELOW_NORMAL	Indică 1 punct sub prioritatea normală a clasei de prioritate.
THREAD_PRIORITY_HIGHEST	Indică 2 puncte deasupra priorității normale a clasei de prioritate.
THREAD_PRIORITY_IDLE	Indică un nivel de prioritate de bază de 1 pentru procesele cu *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS* sau *HIGH_PRIORITY_CLASS* și un nivel de prioritate de bază de 16 pentru procesele cu *REALTIME_PRIORITY_CLASS*.
THREAD_PRIORITY_LOWEST	Indică 2 puncte sub prioritatea normală a clasei de prioritate.
THREAD_PRIORITY_NORMAL	Indică prioritatea normală pentru clasa de prioritate
THREAD_PRIORITY_TIME_CRITICAL	Indică un nivel de prioritate de bază de 15 pentru procesele *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS* sau *HIGH_PRIORITY_CLASS* și un nivel de prioritate de bază de 31 pentru procesele *REALTIME_PRIORITY_CLASS*.

***Tabelul 1400** Valorile nivelurilor de prioritate ale firelor pentru parametrul **nPriority**.*

Dacă funcția *SetThreadPriority* reușește, valoarea returnată este diferită de zero. Dacă funcția eșuează, valoarea returnată este zero. Așa cum ați învățat, fiecare fir are un nivel de prioritate de bază determinat de prioritatea firului și de clasa de prioritate a procesului său.

Sistemul utilizează nivelul de prioritate de bază al tuturor firelor executabile pentru a determina ce fir va primi următoarea secvență de timp CPU. Firele sunt planificate în modul round-rolin la fiecare nivel de prioritate, iar numai atunci când nu se află fire executabile la un nivel superior, e posibilă planificarea firelor de la nivel inferior.

Funcția *SetThreadPriority* permite stabilirea nivelului de prioritate de bază a firului în raport cu clasa de prioritate a procesului său. De exemplu, specificând valoarea *THREAD_PRIORITY_HIGHEST* într-un apel la *SetThreadPriority* pentru un fir al unui proces *IDLE_PRIORITY_CLASS*, nivelul de prioritate de bază a firului se stabilește la 6. Pentru procesele *IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS* și *HIGH_PRIORITY_CLASS* sistemul crește în mod dinamic nivelul de prioritate de bază al firului atunci când apar evenimente importante pentru fir (cum ar fi intrarea în stare de inactivitate a unui alt fir). Procesele *REALTIME_PRIORITY_CLASS* nu beneficiază de această creștere dinamică. Toate firele pornesc inițial de la valoarea *THREAD_PRIORITY_NORMAL*.

Utilizați clasa de prioritate a procesului pentru a face diferențieri între aplicațiile care prezintă urgențe de timp și cele care au cerințe de planificare normale sau sub normal. Utilizați valorile pentru a diferenția prioritățile relative ale activităților unui proces. De exemplu, un fir care gestionează intrările într-o fereastră poate avea o prioritate mai înaltă decât firul care efectuează calcule intensive pentru CPU.

Când manipulați priorități, fiți foarte atenți ca un fir de înaltă prioritate să nu consume tot timpul disponibil de CPU. Un fir cu nivelul de prioritate de bază peste 11 interferează cu operarea normală a sistemului de operare. Utilizarea clasei *REALTIME_PRIORITY_CLASS* poate avea ca efect nesalvarea rezervelor cache pe disc, oprirea mouse-ului și altele.

OBȚINEREA NIVELULUI DE PRIORITATE A FIRULUI CURENT

C/C++1401

Așa cum ați învățat în secțiunea 1400, programele dumneavoastră pot utiliza funcția *SetThreadPriority* pentru schimbarea nivelului de prioritate al firului curent. De multe ori programele vor necesita informații privind nivelul de prioritate al firului curent, de obicei ca pas anterior unui apel la *SetThreadPriority*. Funcția *GetThreadPriority* returnează valoarea de prioritate a unui fir specificat. Această valoare, împreună cu clasa de prioritate a procesului din care face parte firul, determină nivelul priorității de bază a firului. Implementarea funcției *GetThreadPriority* se face potrivit prototipului de mai jos:

```
int GetThreadPriority(
    HANDLE hThread    // identificator pentru fir
);
```

La fel ca în cazul funcției *SetThreadPriority*, parametrul *hThread* identifică firul. Dacă funcția reușește, valoarea returnată va fi nivelul de prioritate a firului. Dacă eșuează, valoarea returnată va fi *THREAD_PRIORITY_ERROR_RETURN*. Pentru informații suplimentare despre erori apelați funcția *GetLastError*. Nivelul de prioritate al firului poate lua una din valorile prezentate în Tabelul 1400.

Fiecare fir are un nivel de prioritate de bază determinat de valoarea de prioritate a firului și de clasa de prioritate a procesului său. Sistemul de operare utilizează nivelul de prioritate de bază al tuturor firelor executabile pentru a determina ce fir va primi următoarea secvență de timp CPU. Firele sunt planificate în modul round-robin la fiecare nivel de prioritate, iar

numai atunci când nu se află fire executabile la un nivel superior, e posibilă planificarea firelor de la nivel inferior.

Observație: *Sub Windows NT, identificatorul trebuie să dețină dreptul de acces THREAD_QUERY_INFORMATION.*

1402 OBȚINEREA CONTEXTULUI UNUI FIR

C/C++

Așa cum ați învățat, Windows stochează informațiile referitoare la fire într-o structură *CONTEXT*. Pe măsură ce programul va manipula fire din ce în ce mai frecvent, veți avea nevoie de informații despre contextul firelor. Funcția *GetThreadContext* obține contextul unui fir specificat. Implementarea funcției *GetThreadContext* se face potrivit prototipului de mai jos:

```
BOOL GetThreadContext(
    HANDLE hThread,      // identificator pentru firul cu context
    LPCONTEXT lpContext // adresa structurii de context
);
```

Parametrul *hThread* conține un identificator deschis la un fir al cărui context urmează a fi preluat. Parametrul *lpContext* indică adresa unei structuri *CONTEXT* care primește contextul corespunzător al firului specificat. Valoarea membrului *ContextFlags* din această structură precizează care porțiune a contextului firului este preluată. Structura *CONTEXT* depinde în mare măsură de calculator. În mod curent, există structuri *CONTEXT* definite pentru procesoare Intel, MIPS, Alpha și PowerPC.

Funcția *GetThreadContext* va fi utilizată pentru preluarea contextului firului specificat. Funcția permite obținerea unui context selectiv bazat pe valoarea membrului *ContextFlags* al structurii *CONTEXT*. Identificatorul firului dat de parametrul *hThread* este de obicei depanat, dar funcția poate opera și fără această depanare. Nu puteți obține un context valid pentru un fir aflat în rulare. Trebuie să utilizați funcția *SuspendThread* înaintea apelării lui *GetThreadContext* pentru a suspenda firul.

Observație: *Sub Windows NT, identificatorul trebuie să dețină dreptul de acces la fir* ***THREAD_GET_CONTEXT***.

1403 ÎNTRERUPEREA ȘI RELUAREA EXECUTĂRII FIRELOR

C/C++

În secțiunile anterioare ați învățat că programele dumneavoastră pot crea fire în stare de suspendare (utilizând indicatorul *CREATE_SUSPENDED*) cu funcțiile *CreateProcess* sau *CreateThread*. Când creați un fir suspendat, sistemul produce obiectul kernel care identifică firul, produce stiva firului și inițializează membrii registre CPU ai structurii *CONTEXT*. Însă crearea funcției conferă obiectului fir un contor inițial de suspendare 1, ceea ce înseamnă că sistemul nu va atribui niciodată timp CPU pentru executarea firului. Pentru a permite firului să intre în execuție, un alt fir trebuie să apeleze funcția *ResumeThread* și să-i transmită identificatorul firului suspendat. Funcția decrementează contorul unui fir suspendat. Când contorul firului suspendat este decrementat la zero, execuția firului este reluată. Implementarea funcției *ResumeThread* se face potrivit prototipului prezentat mai jos:

```
DWORD ResumeThread(HANDLE hThread);
```

Parametrul *hThread* specifică un identificator pentru firul care urmează a fi repornit. Puteți suspenda un fir de mai multe ori, dar va trebui să apelați *ResumeThread* de același număr de ori cu care ați suspendat firul înainte ca el să-și reia execuția.

Funcția *ResumeThread* testează contorul de suspendare al firului respectiv. Când contorul de suspendare este zero, firul nu este la acel moment suspendat. În caz contrar, contorul de suspendare a firului respectiv este decrementat. Dacă valoarea rezultată este zero, execuția firului respectiv este reluată. Dacă valoarea returnată este zero, firul specificat nu a fost suspendat. Dacă valoarea returnată este 1, firul specificat este în continuare suspendat.

Observați că în timpul relatării evenimentelor de depanare, toate firele din procesul de relatare sunt înghețate. Depanatoarele vor utiliza funcția *SuspendThread* și *ResumeThread* pentru a limita setul de fire care se execută într-un proces. Prin suspendarea tuturor firelor dintr-un proces cu excepția celui care raportează un eveniment de depanare, este posibil să se parcurgă un singur „pas" dintr-un singur fir. Celelalte fire nu sunt eliberate printr-o operație continuă dacă sunt suspendate.

Observație: *Sub Windows NT, identificatorul* ***hProcess*** *trebuie să aibe dreptul de acces la fir* ***THREAD_SUSPEND_RESUME***.

SINCRONIZAREA FIRELOR

C/C++1404

Așa cum ați învățat în secțiunile precedente, Windows acceptă executarea de fire multiple. Într-un mediu în care unul sau mai multe fire se execută concurențial, va fi important ca programele să poată sincroniza activitățile diferitelor fire. Sistemul de operare bazat pe Win32 furnizează o serie de obiecte de sincronizare care permit firelor să-și sincronizeze acțiunile cu alte fire. În secțiunea 1405, veți învăța mai mult referitor la obiectele specifice de sincronizare.

În general, un fir se sincronizează cu un altul prin „adormire". Când un fir este adormit, sistemul de operare nu-i mai programează timp CPU, iar firul se oprește din execuție. Înainte însă de a adormi, firul comunică sistemului de operare ce „eveniment special" trebuie să apară (cum ar fi o acționare de tastă, clic de mouse sau încheierea unui algoritm), pentru a intra din nou în execuție.

Sistemul de operare, la rândul lui, rămâne avertizat de cererea firului și urmărește dacă sau când apare evenimentul special. Când evenimentul apare, sistemul de operare alertează firul care devine din nou apt să fie planificat pentru timp CPU. În cele din urmă, CPU va planifica firul și va continua execuția firului, ceea ce înseamnă că firul este acum sincronizat cu apariția evenimentului special.

DEFINIREA CELOR CINCI OBIECTE MAJORE DE SINCRONIZARE

C/C++1405

Așa cum ați învățat în secțiunea 1404, Windows acceptă tipuri diferite de obiecte de sincronizare. În cadrul acestor tipuri, cele mai utilizate sunt: secțiunea critică, excluderea reciprocă, semaforul, evenimentul și contorull de așteptare. Secțiunile următoare vor analiza în detaliu unele din aceste tipuri. Merită totuși să înțelegeți de pe acum definițiile fiecărui tip. Tabelul 1405 prezintă cele cinci tipuri mai importante de obiecte de sincronizare și acțiunile lor.

Tip	Utilizare și acțiune
Secțiunea critică (Critical section)	Secțiunea critică este o secvență mică de cod care cere acces exclusiv la o serie de date partajate, înainte ca respectivul cod să poată începe executarea. Dintre toate obiectele de sincronizare, secțiunile critice sunt cel mai simplu de utilizat. Însă, secțiunile critice trebuie utilizate numai pentru sincronizarea firelor dintr-un singur proces.
Excluderea reciprocă (Mutex)	Excluderea reciprocă se aseamănă mult cu secțiunea critică, însă ea este folosită pentru sincronizarea accesului la date în procese multiple. În plus, excluderile reciproce sunt obiecte kernel, ceea ce înseamnă că programele vor crea efectiv o excludere reciprocă apelând o funcție API, cum este *CreateMutex*.
Semaforul (Semaphore)	Programele vor folosi obiecte semafor pentru contorizarea resurselor. Numai un singur fir poate folosi un semafor pentru contorizarea resurselor disponibile și alocarea acestor resurse. De exemplu, dacă un calculator are trei porturi seriale, puteți crea un semafor cu un contor de resurse egal cu trei. De fiecare dată când firul accesează un port serial, contorul de resurse al semaforului se reduce cu unu, iar de fiecare dată când firul eliberează un port serial, contorul de resurse se incrementează cu unu. În consecință, firele pot apela semaforul și aștepta până când acesta devine disponibil, înainte ca ele să acceseze porturile seriale. Spre deosebire de excluderile reciproce și secțiunile critice, semafoarele nu sunt deținute de fire.
Evenimentul (Event)	Obiectele eveniment sunt cele mai primitive obiecte de sincronizare și sunt foarte diferite de excluderile reciproce și de semafoare. De regulă, programele vor utiliza excluderi reciproce și semafoare pentru controlul accesului la date sau resurse. Programele vor utiliza evenimente pentru semnalarea îndeplinirii unei operații. Programele vor utiliza de cele mai multe ori evenimentele pentru a porni un al doilea fir după ce primul fir efectuează o parte din prelucrarea sa.
Contorul de așteptare (Waitable timer)	Un contor de așteptare este un obiect kernel care se semnalizează periodic pe sine, fie la un moment specificat, fie la intervale regulate. Puteți interpreta un contor de așteptare ca un ceas intern de avertizare al programelor. De exemplu, puteți scrie un program de planificare care avertizează utilizatorul din oră în oră despre noi întâlniri fixate. În loc de a executa o buclă și a aștepta schimbarea orei, programul poate crea un contor de așteptare care semnalează programului schimbările intervenite la fiecare oră. Contoarele de așteptare există numai în Windows NT 4 și în versiuni superioare. Windows 95 nu acceptă acest tip de obiecte de sincronizare.

Tabelul 1405 *Cele cinci tipuri mai importante de obiecte de sincronizare.*

1406 CREAREA UNEI SECȚIUNI CRITICE



Așa cum ați învățat în secțiunea 1405, cel mai simplu tip de sincronizare este secțiunea critică. Secțiunea critică vă permite accesul la o serie specificată de date sau funcții din cadrul programului, asigurând accesul unui singur fir la acele date sau că toate celelalte fire interne ale procesului și-au încheiat prelucrarea înaintea executării secțiuni critice.

Crearea unei secțiuni critice este relativ ușoară. Mai întâi programul trebuie să aloce o structură de date *CRITICAL_SECTION* în procesul ce se execută. Programul trebuie să aloce această structură global, astfel ca diferitele fire din procesul curent să poată accesa instanța *CRITICAL_SECTION* a programului. De regulă, aceasta înseamnă că instanța *CRITICAL_SECTION* va fi o variabilă globală.

După ce programul alocă o structură de date *CRITICAL_SECTION*, el va trebui să efectueze doi pași pentru a crea și intra în secțiunea critică. Programul trebuie pentru început să apeleze funcția *InitializeCriticalSection* pentru inițializarea secțiunii, apoi funcția *EnterCriticalSection* când este pregătit să intre în secțiunea critică. Funcția *EnterCriticalSection* așteaptă ca firul să preia obiectul secțiune critică specificat. Funcția returnează controlul când firul apelant primește proprietatea. Implementarea funcției *EnterCriticalSection* se face potrivit următorului prototip:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Parametrul *lpCriticalSection* indică obiectul secțiune critică. Pentru activarea unei excluderi reciproce la accesul la o resursă partajată, fiecare fir apelează funcția *EnterCriticalSection* sau *TryEnterCriticalSection* pentru a cere proprietatea secțiunii critice înaintea executării oricărei secvențe de cod care accesează resursa protejată. Diferența este că *TryEnterCriticalSection* se returnează imediat, indiferent dacă a obținut proprietatea secțiunii critice, în timp ce *EnterCriticalSection* se blochează până când firul poate prelua proprietatea secțiunii critice. Când a terminat execuția codului protejat, firul utilizează funcția *LeaveCriticalSection* pentru a abandona proprietatea, cedând altui fir posibilitatea de a deveni proprietar și de a accesa resursa partajată. Firul care deține controlul trebuie să apeleze funcția *LeaveCriticalSection* pentru fiecare intrare în secțiunea critică. Firul intră în secțiunea critică de fiecare dată când urmează funcția *EnterCriticalSection* sau *TryEnterCriticalSection*.

După ce un fir a intrat în proprietatea unei secțiuni critice, el poate face apeluri suplimentare la *EnterCriticalSection* sau *TryEnterCriticalSection* fără blocarea execuției sale. Aceasta previne autoblocarea execuției firului în timpul așteptării unei secțiuni critice deja deținute.

Orice fir al procesului poate utiliza funcția *DeleteCriticalSection* pentru eliberarea resurselor sistemului alocate la inițializarea obiectului secțiune critică. După apelarea acestei funcții, obiectul secțiune critică nu mai poate fi utilizat pentru sincronizare.

Observație: *Deși membrii structurii de date* ***CRITICAL_SECTION*** *sunt definiți în fișierul antet* ***winbase.h****, programul dumneavoastră nu trebuie să acceseze membrii structurii pentru că Windows gestionează intern informațiile, iar modificările membrilor pot cauza erori de sistem fatale.*

UTILIZAREA UNEI SECȚIUNI CRITICE SIMPLE

C/C++1407

Așa cum ați învățat în secțiunea 1406, crearea și utilizarea unei secțiuni critice este un proces cu minimum trei etape: crearea secțiunii, inițializarea secțiunii și intrarea în secțiune. Înainte de a sincroniza fire cu o secțiune critică, trebuie să inițializați secțiunea critică, prin transmiterea adresei structurii de date *CRITICAL_SECTION* ca singur parametru. Când ați ajuns la începutul secțiunii critice, programul trebuie să apeleze fie funcția *EnterCriticalSection*, fie *TryEnterCriticalSection*, din nou cu transmiterea adresei structurii de date *CRITICAL_SECTION*

ca singur parametru. După ce firul s-a sincronizat, secţiunea critică rămâne până când programul părăseşte sau şterge secţiunea critică.

Pentru a înţelege mai bine prelucrarea efectuată de programul dumneavoastră când gestionaţi secţiuni critice, să analizăm programul *Crit_Section.cpp* conţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *Crit_Section* utilizează o secţiune critică cu o instrucţiune de adormire pentru cinci secunde pentru a permite numai unui singur fir să execute codul critic la un moment dat. De fiecare dată când utilizatorul selectează opţiunea *Test!* programul creează un alt fir care, în schimb, aşteaptă să intre în secţiunea critică. Codul operativ al programului *Crit_Section.cpp* se află în funcţiile *ChildThreadProc* şi *WndProc*.

1408 UTILIZAREA FUNCŢIEI WAITFORSINGLEOBJECT PENTRU SINCRONIZAREA A DOUĂ FIRE



Aşa cum aţi învăţat, multe activităţi de sincronizare se desfăşoară în jurul aşteptării unuia sau mai multor fire, înaintea continuării execuţiei firului curent. Când firul curent aşteaptă ca CPU să se întoarcă de la un alt fir, programul dumneavoastră trebuie să utilizeze funcţia *WaitForSingleObject* care se returnează când are loc unul din următoarele evenimente:

- Obiectul specificat este în stare de semnalizare
- Intervalul de suspendare a fost depăşit

Programele dumneavoastră vor utiliza funcţia *WaitForSingleObject* după următorul prototip:

```
DWORD WaitForSingleObject(
    HANDLE hHandle,  // identificatorul obiectului de asteptat
    DWORD dwMilliseconds  // interval de intrerupere in
                          // milisecunde
);
```

Funcţia *WaitForSingleObject* acceptă doi parametri, *hHandle* şi *dwMilliseconds*. Parametrul *hHandle* identifică obiectul pentru care aşteaptă funcţia. Parametrul *dwMilliseconds* specifică intervalul de suspendare în milisecunde. Funcţia se returnează dacă intervalul se epuizează, chiar dacă starea obiectului nu este semnalizată. Dacă *dwMilliseconds* este zero, funcţia verifică starea obiectului şi se returnează imediat. Dacă *dwMilliseconds* este *INFINITE* intervalul de suspendare nu se epuizează niciodată. Tabelul 1408.1 prezintă o listă a tipurilor de obiecte ai căror identificatori pot fi specificaţi (adică tipuri de obiecte pe care funcţia *WaitForSingleObject* le aşteaptă).

Obiect	Descriere
Notificare pentru schimbare (Change notification)	Funcţia *FindFirstChangeNotification* returnează acest identificator. O notificare a schimbării stării obiectului este semnalizată când un anumit tip de schimbare intervine în directorul respectiv sau în arborele său.
Intrare de la consolă (Console input)	Identificatorul este returnat de către funcţia *CreateFile* când este specificată valoarea *CONIN$* sau de către funcţia *GetStdHandle*. Starea obiectului este semnalizată când există o intrare necitită în bufferul de intrare de la consolă şi este nesemnalizată când bufferul de intrare este gol.

Obiect	Descriere
Eveniment (Event)	Funcțiile *CreateEvent* sau *OpenEvent* returnează acest identificator. Starea obiectului eveniment este stabilită explicit ca fiind semnalizată prin funcțiile *SetEvent* sau *PulseEvent*. Starea obiectului eveniment cu reinițializare manuală trebuie adusă pe nesemnalizat prin funcția *ResetEvent*. Pentru un obiect eveniment cu auto-reset, funcția de așteptare reinițializează starea obiectului pe nesemnalizat înainte de returnare. Obiectele eveniment sunt de asemenea utilizate în operațiuni suprapuse, în care starea este stabilită de sistem.
Excludere mutuală (Mutex)	Funcțiile *CreateMutex* sau *OpenMutex* returnează acest identificator. Starea unui obiect de excludere reciprocă (mutex) este semnalizată când obiectul nu este deținut de nici un fir. Funcția de așteptare cere preluarea în proprietate a excluderii reciproce pentru firul apelant, schimbând starea excluderii reciproce pe nesemnalizat când s-a acordat proprietatea.
Proces (Process)	Funcțiile *CreateProcess* sau *OpenProcess* returnează acest identificator. Starea obiectului proces este semnalizată când procesul este încheiat.
Semafor (Semaphore)	Funcțiile *CreateSemaphore* sau *OpenSemaphore* returnează acest identificator. Obiectul semafor menține un contor între zero și o oarecare valoare maximă. Starea lui este semnalizată când contorul său este mai mare ca zero și nesemnalizată când contorul este zero. Dacă starea curentă este semnalizată, funcția de așteptare decrementează contorul cu unu.
Fir de execuție (Thread)	Funcțiile *CreateProcess*, *CreateThread* sau *CreateRemoteThread* returnează acest identificator. Starea unui obiect fir este semnalizată când firul este încheiat.
Contor (Timer)	Funcțiile *CreateWaitableTimer* sau *OpenWaitableTimer* returnează acest identificator. Activați contorul prin apelarea funcției *SetWaitableTimer*. Starea unui contor este semnalizată când a parcurs intervalul stabilit. Puteți dezactiva contorul prin apelarea funcției *CancelWaitableTimer*.

Tabelul 1408.1 *Obiectele așteptate de funcția* ***WaitForSingleObject***.

Observație: *În Windows NT, identificatorul trebuie să aibe accesul de tip* ***SYNCHRONIZE***.

Dacă funcția *WaitForSingleObject* eșuează, valoarea returnată va fi *WAIT_FAILED*. Dacă funcția *WaitForSingleObject* reușește, valoarea returnată indică evenimentul care a determinat funcția să returneze. Valoarea returnată pentru invocarea reușită a funcției este oricare din valorile prezentate în Tabelul 1408.2.

Valoare	Semnificație
WAIT_ABANDONED	Obiectul specificat este un obiect de excludere reciprocă (mutex) care nu a fost eliberat de firul care a posedat obiectul înainte ca firul proprietar să se fi terminat. Proprietatea excluderii reciproce este preluată de firul apelant, iar excluderea reciprocă este făcută nesemnalizată.
WAIT_OBJECT_0	Starea obiectului specificat este semnalizată.
WAIT_TIMEOUT	Intervalul de suspendare s-a încheiat și starea obiectului este nesemnalizată.

***Tabelul 1408.2** Valorile returnate în cazul reușitei funcției **WaitForSingleObject**.*

Funcția *WaitForSingleObject* testează starea curentă a obiectului respectiv. Dacă starea obiectului este nesemnalizată, firul apelant intră într-o stare de așteptare. Firul consumă foarte puțin timp CPU în timp ce așteaptă ca starea obiectului să devină semnalizată sau ca intervalul de întrerupere să se epuizeze. Înaintea returnării, o funcție de așteptare modifică starea unor tipuri de obiecte de sincronizare. Modificările apar numai pentru obiectul sau obiectele a căror stare semnalizată a determinat ca funcția să se returneze. De exemplu, funcția de așteptare decrementează un obiect semafor cu 1.

Trebuie să aveți grijă atunci când folosiți funcțiile de așteptare împreună cu schimbul dinamic de date (DDE). Dacă un fir a creat o fereastră, el va trebui să prelucreze mesaje. Schimbul dinamic de date trimite mesaje tuturor ferestrelor din sistem. Dacă aveți un fir care folosește o funcție de așteptare fără interval de suspendare, sistemul se va bloca. În consecință, dacă aveți un fir care creează ferestre, utilizați *MsgWaitForMultipleObjects* sau *MsgWaitForMultipleObjectsEx* și nu *WaitForSingleObject*.

Pentru a înțelege mai bine prelucrarea efectuată de funcția *WaitForSingleObject*, să analizăm programul *Wait_Events.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. De fiecare dată când utilizatorul selectează opțiunea *Test!*, programul *Wait_Events* pornește un fir care așteaptă un eveniment, adoarme și apoi părăsește evenimentul. Utilizarea evenimentelor auto-reset (evenimente care își schimbă automat indicatoarele de semnalizare) forțează accesul firului în ordine serială, pentru că evenimentele își reinițializează automat valoarea pe nesemnalizat atunci când primul fir în așteptare primește obiectul. Programul *Wait_Events.cpp* își efectuează prelucrarea operativă în funcțiile *ChildThreadProc* și *WndProc*.

1409 UTILIZAREA FUNCȚIEI WAITFORMULTIPLEOBJECTS PENTRU SINCRONIZAREA FIRELOR MULTIPLE



În secțiunea 1408 ați învățat cum să utilizați funcția *WaitForSingleObject* pentru sincronizarea unui fir cu un singur eveniment de tip auto-reset. De cela mai multe ori însă, programele dumneavoastră vor cere ca prelucrarea să continue numai când apare unul sau mai multe obiecte dintr-un anumit set. În aceste cazuri, veți utiliza funcția *WaitForMultipleObjects*. Această funcție se returnează când intervine una din următoarele situații:

- Unul sau toate obiectele respective sunt în stare semnalizată.
- Timpul de suspendare s-a epuizat.

Implementarea funcției *WaitForMultipleObjects* se face potrivit următorului prototip:

```
DWORD WaitForMultipleObjects(
    DWORD nCount,        // numar de identificatori in matricea cu
                         // identificatori de obiecte
    CONST HANDLE *lpHandles, // pointer la matricea cu
                             // identificatori de obiecte
    BOOL bWaitAll,           // indicator de asteptare
    DWORD dwMilliseconds // interval intrerupere in milisecunde
);
```

Funcția *WaitForMultipleObjects* acceptă parametrii prezentați în Tabelul 1409.1.

Parametru	Descriere
nCount	Specifică numărul de identificatori de obiecte în matricea indicată de *lpHandles*. Numărul maxim de identificatori de obiecte este *MAXIMUM_WAIT_OBJECTS* (o constantă definită de sistem care variază de la o instalare la alta).
lpHandles	Indică o matrice cu identificatori de obiecte. Tabelul 1408.1 enumeră tipurile de obiecte ai căror identificatori îi puteți specifica.
OBSERVAȚIE: *Sub Windows NT, identificatorii trebuie să dețină dreptul de acces SYNCHRONIZE.*	
bWaitAll	Specifică tipul de așteptare. Dacă acesta este TRUE, funcția se returnează când starea tuturor obiectelor din matricea *lpHandles* este semnalizată. Dacă parametrul este FALSE, funcția se returnează când oricare din obiecte este semnalizat. În acest ultim caz, valoarea returnată indică obiectul a cărui stare a determinat funcția să se returneze.
dwMilliseconds	Specifică intervalul de întrerupere, în milisecunde. Funcția se returnează dacă intervalul s-a scurs, chiar dacă nu sunt întrunite condițiile specificate de parametrul *bWaitAll*. Dacă *dwMilliseconds* este zero, funcția testează starea obiectelor specificate și se returnează imediat. Dacă *dwMilliseconds* este *INFINITE* intervalul de suspendare a funcției nu se epuizează niciodată.

Tabelul 1409.1 *Parametrii funcției* ***WaitForMultipleObjects*.**

După invocare, funcția *WaitForMultipleObjects* se va returna în funcție de eșec, reușită și timpul de suspendare. Dacă *WaitForMultipleObjects* eșuează, valoarea returnată va fi *WAIT_FAILED*, dacă *WaitForMultipleObjects* reușește, valoarea returnată va indica evenimentul care a determinat ca funcția să se returneze, fiind una din valorile enumerate în Tabelul 1408.2.

Funcția *WaitForMultipleObjects* determină dacă unul sau mai multe obiecte așteptate de fir a întrunit criteriile de așteptare. Dacă nici unul din obiectele așteptate de fir nu a întrunit criteriile de așteptare, firul apelant intră într-o stare de așteptare, pentru întrunirea criteriilor, cu un consum minim de timp CPU. Când *bWaitAll* este TRUE, operația de așteptare a funcției va fi încheiată când starea tuturor obiectelor este semnalizată. Parametrul *bWaitAll* nu modifică stările obiectelor spoecificate până când stările tuturor obiectelor nu sunt semnalizate. De exemplu, o excludere reciprocă poate fi semnalizată, dar firul nu preia proprietatea până când starea celuilalt obiect nu este de asemenea semnalizată. Între timp, alte câteva fire vor prelua proprietatea excluderii reciproce, fixându-i starea pe nesemnalizat. Înainte de a se returna, o funcție de așteptare modifică starea unor tipuri de obiecte de sincronizare. Modificările apar numai pentru obiectul sau obiectele a căror stare semnalizată a determinat funcția să se returneze. De exemplu, contorul unui obiect semafor este decre-

mentat cu unu de către un alt fir sau procese. Funcția *WaitForMultipleObjects* poate specifica în matricea *lpHandle* unul sau mai mulți identificatori de obiecte cu tipurile enumerate în Tabelul 1408.1.

Ca și în cazul funcției *WaitForSingleObject*, trebuie să aveți grijă atunci când folosiți funcțiile de așteptare împreună cu schimbul dinamic de date. Dacă un fir a creat vreo fereastră, el va trebui să prelucreze mesaje. Schimbul dinamic de date trimite mesaje tuturor ferestrelor din sistem. Dacă aveți un fir care folosește o funcție de așteptare fără interval de suspendare, sistemul se va bloca. În consecință, dacă aveți un fir care creează ferestre, utilizați funcția *MsgWaitForMultipleObjects* sau *MsgWaitForMultipleObjectsEx*, și nu *WaitForMultipleObjects*.

1410 CREAREA UNEI EXCLUDERI RECIPROCE

C/C++

Așa cum ați învățat, excluderile reciproce (*mutex*) sunt asemănătoare secțiunilor critice cu excepția faptului că programele dumneavoastră pot utiliza excluderile reciproce și pentru sincronizarea accesului la date pentru mai multe obiecte, nu numai pentru unul. Pentru utilizarea unei excluderi reciproce, programul trebuie mai întâi să o creeze cu funcția *CreateMutex*. Implementarea funcției *CreateMutex* se face conform prototipului descris mai jos:

```
HANDLE CreateMutex(
     LPSECURITY_ATTRIBUTES lpMutexAttributes, // atributele de
                                              // securitate
     BOOL bInitialOwner, // indicator pentru proprietatea
                         // initiala
     LPCTSTR lpName // pointer la numele obiectului cu
                    // excludere reciproca
);
```

Parametrul *lpMutexAttributes* este un pointer la o structură *SECURITY_ATTRIBUTES* care determină dacă identificatorul returnat poate fi moștenit de către procesele copil. Dacă *lpMutexAttributes* este NULL, identificatorul nu poate fi moștenit. Sub Windows NT, membrul *lpSecurityDescriptor* al structurii specifică un descriptor de securitate pentru noua excludere reciprocă. Dacă *lpMutexAttributes* este NULL, excluderea reciprocă preia descriptorul de securitate implicit. Sub Windows 95, membrul *lpSecurityDescriptor* este ignorat. Parametrul *bInitialOwner* specifică proprietarul inițial al obiectului excludere reciprocă. Parametrul *bInitialOwner* specifică proprietarul inițial al obiectului mutex. Dacă este TRUE, firul apelant cere imediat proprietatea excluderii reciproce. Altfel, nici un fir nu deține obiectul mutex. Parametrul *lpName* indică un șir terminat cu NULL care specifică numele excluderii reciproce. Numele este limitat la numărul de caractere dat de *MAX_PATH* și poate conține orice caracter cu excepția caracterului *backslash(\)*. Compararea numelui este indiferentă la majuscule-minuscule.

Dacă *lpName* este identic cu numele unui obiect cu excludere reciprocă existent, funcția cere dreptul de acces *MUTEX_ALL_ACCESS* la obiectul respectiv. În acest caz, parametrul *bInitialOwner* este ignorat deoarece a fost deja stabilit de procesul apelant. Dacă parametrul *lpMutexAttributes* nu este NULL, *CreateMutex* determină dacă identificatorul poate fi moștenit, dar membrul său descriptor de securitate este ignorat. Dacă *lpName* este NULL, funcția *CreateMutex* produce un obiect cu excludere reciprocă fără nume. Dacă funcția *lpName* are același nume cu al unui obiect eveniment, semafor sau al unui obiect fișier de mapare existente, funcția eșuează, iar funcția *GetLastError* returnează *ERROR_INVALID_HANDLE*.

Funcția eșuează deoarece obiectele eveniment, cu excludere reciprocă, semafor și fișier de mapare partajează același spațiu de nume.

Identificatorul returnat de *CreateMutex* are dreptul de acces *MUTEX_ALL_ACCESS* la noul obiect cu excludere reciprocă și poate fi utilizat de orice funcție care cere un identificator la un obiect cu excludere reciprocă. Orice fir al procesului apelant poate specifica identificatorul obiectului cu excludere reciprocă într-un apel la o funcție de așteptare (cum ar fi *WaitForSingleObject* și *WaitForMultipleObjects*). Funcțiile de așteptare cu un singur obiect se returnează când starea obiectului respectiv este semnalizată. Funcțiile de așteptare multi-obiect se returnează când unul sau mai multe obiecte specificate sunt semnalizate. Când o funcție de așteptare se returnează, firul care așteaptă este lăsat să-și continue execuția.

Starea unui obiect cu excludere reciprocă este semnalizată când nu este în proprietatea nici unui fir. Firul care îl creează poate utiliza indicatorul *bInitialOwner* pentru a cere imediat proprietatea obiectului cu excludere reciprocă. Altfel, firul trebuie să utilizeze una din funcțiile de așteptare pentru a cere proprietatea. Când starea excluderii este semnalizată, un fir în așteptare obține proprietatea, starea excluderii reciproce se modifică pe nesemnalizat, iar funcția de așteptare se returnează. Numai un fir poate deține o excludere reciprocă în orice moment. Firul proprietar utilizează funcția *ReleaseMutex* pentru înlăturarea proprietății. Firul care deține o excludere reciprocă poate specifica aceeași excludere reciprocă în apeluri repetate de funcții de așteptare, fără blocarea execuției. De regulă, un fir nu va aștepta în mod repetat pentru aceeași excludere reciprocă, dar acest mecanism previne ca firul să se blocheze singur în timpul așteptării unei excluderi reciproce deținute deja. Însă, pentru a i se lua proprietatea, firul trebuie să apeleze o dată funcția *ReleaseMutex*, de fiecare dată când excluderea reciprocă satisface o așteptare.

Două sau mai multe procese pot apela funcția *CreateMutex* pentru crearea unei excluderi reciproce cu același nume. Primul proces creează o excludere reciprocă, iar procesele următoare deschid un identificator către acea excludere reciprocă. Procesele multiple pot utiliza funcția *CreateMutex* pentru a le permite să obțină identificatori ai aceluiași obiect mutex, sau țind utilizatorul de a verifica faptul că procesul care creează mai întâi a pornit excluderea reciprocă. Când utilizați această tehnică, trebuie să stabiliți indicatorul *bInitialOwner* pe FALSE; în caz contrar, poate fi dificil de aflat care proces deține inițial proprietatea. Procesele multiple pot avea identificatori la același obiect cu excludere reciprocă, permițând programelor să utilizeze obiectul mutex pentru sincronizarea între procese. Sunt disponibile următoarele mecanisme de partajare a obiectului:

- Un proces copil creat de funcția *CreateProcess* poate moșteni un identificator de obiect cu excludere reciprocă, în cazul în care parametrul *lpMutexAttributes* al funcției *CreateMutex* activează moștenirea.
- Un proces poate specifica identificatorul la obiectul cu excludere reciprocă printr-un apel la funcția *DuplicateHandle* pentru crearea unui identificator duplicat ce poate fi utilizat de către un alt proces.
- Un proces poate specifica identificatorul obiectului cu excludere reciprocă printr-un apel la funcțiile *OpenMutex* sau *CreateMutex*.

Apelați funcția *CloseHandle* pentru închiderea identificatorului. Sistemul închide automat identificatorul când procesul este terminat. Obiectul cu excludere reciprocă este distrus când ultimul său identificator a fost închis.

1411 UN EXEMPLU DE UTILIZARE A UNEI EXCLUDERI RECIPROCE

C/C++

Așa cum ați învățat în secțiunea 1410, programul dumneavoastră poate crea ușor obiecte de excludere reciprocă pentru sincronizarea unui fir în cadrul mai multor procese. În secțiunea 1410 a fost prezentat procesul de bază al creării unui obiect de excludere reciprocă. Însă, după de ați creat o excludere reciprocă, programul trebuie să obțină un identificator la acea excludere reciprocă înainte de a-l utiliza în propriul său cod. Așa cum ați învățat, un al doilea proces poate obține un identificator la o excludere reciprocă prin invocarea instrucțiunii *CreateMutex* cu aceeași denumire de excludere reciprocă ca în primul proces. În același scop se poate utiliza funcția *OpenMutex*. Această funcție returnează identificatorul unui obiect cu excludere reciprocă creat anterior. Implementarea funcției *OpenMutex* se face conform prototipului descris mai jos:

```
HANDLE OpenMutex(
      DWORD dwDesiredAccess,    // indicator de acces
      BOOL bInheritHandle,      // indicator de mostenire
      LPCTSTR lpName            // pointer la un nume de obiect cu
                                // excludere reciproca
);
```

Parametrul *dwDesiredAccess* specifică accesul solicitat la obiectul cu excludere reciprocă. Pentru sistemele care acceptă securitatea obiectelor (cum sunt instalările de siguranță pentru Windows NT), parametrul va eșua dacă descriptorul de securitate al obiectului specificat nu permite accesul solicitat de procesul apelant. Parametrul *dwDesiredAccess* poate fi orice combinație a valorilor: *MUTEX_ALL_ACCESS* care specifică toate indicatoarele de acces posibile pentru obiectele cu excludere reciprocă, *SYNCHRONIZE*, acceptată numai de Windows NT și care permite utilizarea identificatorului mutex în oricare din funcțiile de așteptare pentru preluarea proprietății excluderii reciproce, sau în funcția *ReleaseMutex* pentru cedarea proprietății, sau în amândouă.

Parametrul *bInheritHandle* specifică dacă identificatorul returnat poate fi moștenit, dacă valoarea este TRUE, procesul anterior creat de funcția *CreateProcess* poate moșteni identificatorul; altfel el nu poate fi moștenit. Parametrul *lpName* indică un șir terminat cu NULL care denumește excluderea reciprocă care va fi deschisă cu *OpenMutex*. Compararea numelor va fi indiferentă la majuscule-minuscule.

Funcția *OpenMutex* permite proceselor multiple deschiderea identificatoarelor aceluiași obiect cu excludere reciprocă. Funcția reușește numai dacă un anumit proces a creat deja o excludere reciprocă utilizând funcția *CreateMutex*. Procesul apelant poate utiliza identificatorul returnat în orice funcție care cere un identificator de obiect cu excludere reciprocă, cum ar fi o funcție de așteptare, cu limitările de acces specificate de parametrul *dwDesiredAccess*.

Programul dumneavoastră poate utiliza funcția *DuplicateHandle* pentru a duplica un indentificator și funcția *CloseHandle* pentru închiderea identificatorului. Sistemul închide automat identificatorul când procesul este terminat. Sistemul de operare distruge obiectul cu excludere reciprocă când sistemul a închis ultimul identificator al obiectului cu excludere reciprocă.

Pentru a înțelege mai bine prelucrările efectuate de program cu obiectele cu excludere reciprocă, să analizăm programul *Simple_Mutex.cpp* conținut pe CD-ROM-ul care însoțește

cartea de față. Programul *Simple_Mutex.cpp* creează un obiect cu excludere reciprocă simplu. Când utilizatorul selectează din meniu opțiunea *Test!*, programul pornește un fir care așteaptă pentru accesul la o excludere reciprocă, intră în adormire și apoi eliberează excluderea. Dacă utilizatorul selectează opțiunea *Test!* de mai multe ori, programul va demonstra conceptul de *competiție* al excluderii reciproce. Aceasta înseamnă că excluderea reciprocă va forța firele ulterioare să aștepte până când fiecare fir precedent își termină execuția. Deoarece programul *Simple_Mutex.cpp* creează un fir copil în cadrul funcției *WndProc*, majoritatea prelucrărilor operative ale programului intervin în cadrul funcției *ChildThreadProc*.

UTILIZAREA SEMAFOARELOR

C/C++1412

Așa cum ați învățat, majoritatea semafoarelor utilizează un contor pentru sincronizare. În cadrul programelor dumneavoastră veți utiliza semafoarele pentru limitarea accesului la un obiect, un fragment de cod sau altă sursă limitată. Când cereți un semafor, veți comunica semaforului câte accese trebuie să permită și numărul inițial de accese. Pentru crearea unui semafor veți utiliza funcția *CreateSemaphor*. Implementarea funcției *CreateSemaphor* se face conform prototipului descris mai jos:

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // atributele
                                   // de securitate
    LONG lInitialCount, // contor initial
    LONG lMaximumCount, // contor maxim
    LPCTSTR lpName       // pointer la numele obiectului semafor
);
```

Funcția *CreateSemaphor* acceptă parametrii prezentați în Tabelul 1412.

Parametru	Descriere
lpSemaphoreAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care stabilește dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpSemaphoreAttributes* este NULL, identificatorul nu poate fi moștenit. Sub Windows NT, membrul *lpSecurityDescriptor* al structurii specifică un descriptor de securitate pentru noul semafor. Dacă *lpSemaphoreAttributes* este NULL, semaforul obține un descriptor de securitate implicit. Sub Windows 95, acest membru este ignorat.
lInitialCount	Specifică un contor inițial pentru obiectul semafor. Această valoare trebuie să fie mai mare sau egală cu zero și mai mică sau egală cu *lMaximumCount*. Starea semaforului este semnalizată când contorul său este mai mare decât zero și nesemnalizată când este zero. Contorul este decrementat cu unu de fiecare dată când o funcție de așteptare eliberează un fir care aștepta semaforul. Contorul crește cu un anumit număr prin apelarea funcției *ReleaseSemaphore*.
lMaximumCount	Conține contorul maxim pentru obiectul semafor. Această valoare trebuie să fie mai mare ca zero.

(continuare)

Parametru	Descriere
lpName	Indică un şir terminat cu NULL care specifică numele obiectului semafor. Numele este limitat la un număr de caractere egal cu *MAX_PATH* şi poate conţine orice caracter cu excepţia caracterului *backslash(\)*.Compararea numelor va fi indiferentă la majuscule-minuscule. Dacă *lpName* este identic cu numele unui obiect semafor existent, funcţia cere dreptul de acces *SEMAPHORE_ALL_ACCESS* la respectivul obiect. În acest caz, parametrii *lInitialCount* şi *MaximumCount* sunt ignoraţi pentru că au fost deja stabiliţi de procesul care a creat semaforul. Dacă parametrul *lpSemaphoreAttributes* nu este NULL, el stabileşte dacă identificatorul poate fi moştenit, dar membrul său descriptor de securitate este ignorat. Dacă *lpName* este NULL obiectul semafor este creat fără denumire. Dacă *lpName* este identic cu numele unui eveniment existent, excludere reciprocă sau fişier de mapare, funcţia eşuează, iar funcţia *GetLastError* returnează constanta *ERROR_INVALID_HANDLE*. Aceasta apare datorită faptului că obiectele eveniment, excludere reciprocă, semafor şi fişier de mapare partajează acelaşi spaţiu de nume.

***Tabelul 1412** Parametrii funcţiei **CreateSemaphor**.*

Dacă funcţia *CreateSemaphor* reuşeşte, valoarea returnată este un identificator pentru obiectul semafor, dacă obiectul semafor denumit există anterior apelului funcţiei, funcţia *GetLastErrori* returnează eroarea *ERROR_ALREADY_EXISTS*. Dacă funcţia eşuează, valoarea returnată va fi NULL.

Identificatorul returnat de *CreateSemaphore* are accesul *SEMAPHORE_ALL_ACCESS* către noul obiect semafor şi poate fi utilizat în orice funcţie care solicită un identificator de obiect semafor. Orice fir al procesului apelant poate specifica identificatorului de obiect semafor într-un apel la o funcţie de aşteptare. Funcţiile de aşteptare cu un singur obiect se returnează când starea obiectului specificat este semnalizată. Funcţiilor de aşteptare multi-obiect li se poate cere să se returneze fie atunci când oricare sau toate obiectele specificate sunt semnalizate. Când o funcţie de aşteptare se returnează, firul care aşteaptă este lăsat să-şi continue execuţia.

Starea semaforului este semnalizată când contorul său este mai mare decât zero şi nesemnalizată când este zero. Parametrul *lInitialCount* conţine contorul iniţial. De fiecare dată când un fir în aşteptare este eliberat datorită stării semnalizate a semaforului, contorul semaforului este decrementat cu unu. Utilizaţi funcţia *ReleaseSemaphore* pentru incrementarea cu o anumită valoare a contorului semaforului. Contorul nu poate fi niciodată mai mic decât zero sau mai mare decât valoarea specificată în parametrul *lMaximumCount*.

Procesele multiple pot avea identificatori pentru acelaşi obiect semafor, activând utilizarea obiectului pentru sincronizarea între procese. Sunt disponibile următoarele mecanisme de partajare a obiectului:

- Un proces copil creat de funcţia *CreateProcess* poate moşteni un identificator pentru un obiect semafor, în cazul în care parametrul *lpSemaphoreAttributes* al funcţiei *CreateSemaphore* activează moştenirea.

- Un proces poate specifica identificatorul pentru obiectul semafor printr-un apel la funcţia *DuplicateHandle* pentru crearea unui identificator duplicat ce poate fi utilizat de către un alt proces.
- Un proces poate specifica denumirea obiectului semafor printr-un apel la funcţiile *OpenSemaphore* sau *CreateSemaphore*.

pelaţi funcţia *CloseHandle* pentru închiderea identificatorului. Sistemul închide automat entificatorul când procesul este terminat. Obiectul semafor este distrus când ultimul său entificator a fost închis.

şa cum aţi învăţat în secţiunea 1411, programele dumneavoastră utilizează o excludere ciprocă prin crearea unei excluderi reciproce, prin deschiderea şi apoi eliberarea sa. În zul semaforului, veţi parcurge etape asemănătoare. Pentru a înţelege mai bine prelucrările ectuate de program cu obiectele semafor, să analizăm programul *Create_Semaphore.cpp* onţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *Create_Semaphore.cpp* ilizează obiecte semafor pentru ca numai patru fire să poată, la un moment dat, să execute ocedura firului copil. De fiecare dată când utilizatorul selectează din meniu opţiunea *Test!*, ogramul încearcă să creeze un nou fir. Însă, procedura firului se asigură că există spaţiu sponibil pentru fir (până la patru) înainte de a permite firului efectuarea prelucrării. Dacă u există suficiente resurse pentru semafoare, procedura firului aşteaptă până când un oces devine disponibil. Prelucrarea operativă intervine în funcţia *ChildThreadProc* a ogramului *Create_semaphore.cpp*.

UTILIZAREA UNUI PROCESOR DE EVENIMENT SIMPLU

C/C++1413

venimentele sunt cea mai primitivă formă de sincronizare a obiectelor. Programele umneavoastră vor utiliza, în general, un eveniment pentru a semnala unuia sau mai multor re că operaţia s-a îndeplinit. La fel ca în cazul excluderii reciproce şi al semaforului, rogramul dumneavoastră va crea un eveniment prin apelarea funcţiei *CreateEvent*. Implementarea funcţiei *CreateEvent* se face potrivit următorului prototip:

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, // atributele de
                                             // securitate
    BOOL bManualReset,   // indicator pt. eveniment de
                         // initializare manuala
    BOOL bInitialState,  // indicator pt. starea initala
    LPCTSTR lpName       // pointer la numele obiectului
                         // eveniment
);
```

uncţia *CreateEvent* acceptă parametrii descrişi în Tabelul 1413.

Parametru	Descriere
lpEventAttributes	Indică o structură *SECURITY_ATTRIBUTES* care stabilește dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpEventAttributes* este NULL, identificatorul nu poate fi moștenit. Sub Windows NT, membrul *lpSecurityDescriptor* al structurii specifică un descriptor de securitate pentru noul eveniment. Dacă *lpEventAttributes* este NULL, evenimentul obține un descriptor de securitate implicit. Sub Windows 95, această funcție ignoră membrul *lpSecurityDescriptor*.
bManualReset	Specifică dacă a fost creat un eveniment prin inițializare manuală sau prin auto-reset. Dacă parametrul este TRUE, trebuie să utilizați funcția *ResetEvent* pentru inițializarea manuală a stării pe nesemnalizat. Dacă este FALSE, Windows inițializează automat starea pe nesemnalizat după ce a fost eliberat un singur fir în așteptare.
bInitialState	Specifică starea inițială a unui obiect eveniment. Dacă parametrul este TRUE, starea inițială este semnalizată; în caz contrar, este nesemnalizată.
lpName	Indică un șir terminat în NULL care specifică numele unui obiect eveniment. Numele este limitat la un număr de caractere *MAX_PATH* și poate conține orice caracter cu excepția caracterului separator de cale – *backslash(\)*. Compararea numelor va fi indiferentă la majuscule-minuscule. Dacă *lpName* este identic cu unui obiect eveniment denumit, funcția cere dreptul de acces *EVENT_ALL_ACCESS* la respectivul obiect. În acest caz, parametrii *bManualReset* și *bInitialState* sunt ignorați pentru că au fost deja stabiliți de procesul care a creat evenimentul. Dacă parametrul *lpEventAttributes* nu este NULL, el stabilește dacă identificatorul poate fi moștenit, dar membrul său descriptor de securitate este ignorat. Dacă *lpName* este NULL, obiectul eveniment este creat fără denumire. Dacă *lpName* este identic cu numele unui semafor existent, excludere reciprocă sau fișier de mapare, funcția eșuează, iar funcția *GetLastError* returnează constanta *ERROR_INVALID_HANDLE*. Această eroare apare pentru că obiectele eveniment, excludere reciprocă, semafor și fișier de mapare partajează același spațiu de nume.

***Tabelul 1413** Parametrii funcției **CreateEvent**.*

Dacă funcția *CreateEvent* reușește, valoarea returnată este un identificator la obiectul eveniment. Dacă obiectul eveniment denumit există înainte de apelul funcției, funcția *GetLastError* returnează eroarea *ERROR_ALREADY_EXISTS*. Dacă funcția eșuează, valoarea returnată va fi NULL.

Identificatorul returnat de *CreateEvent* are dreptul de acces *EVENT_ALL_ACCESS* către noul obiect eveniment și poate fi utilizat în orice funcție care solicită un identificator la un obiect eveniment. Orice fir al procesului apelant poate specifica identificatorul obiectului eveniment printr-un apel la o funcție de așteptare. Funcțiile de așteptare uni-obiect se returnează când starea obiectului specificat este semnalizată. Funcțiile de așteptare multi-obiect pot fi solicitate să se returneze atunci când oricare sau toate obiectele specificate sunt semnalizate. Când o funcție de așteptare se returnează, firul apelant este lăsat să-și continue execuția.

Starea inițială a obiectului eveniment este specificată de parametrul *bInitialState*. Utilizați funcția *SetEvent* pentru stabilirea stării obiectului eveniment pe semnalizat. Utilizați funcția

ResetEvent pentru inițializarea stării unui obiect eveniment pe nesemnalizat. Când starea unui obiect eveniment inițializată manual este semnalizată, ea rămâne semnalizată până când este explicit inițializată pe nesemnalizat de către funcția *ResetEvent.* Programul dumneavoastră poate elibera orice număr de fire de așteptare sau fire care ulterior încep operații de așteptare pentru obiectul eveniment specificat, dacă starea obiectului este semnalizată.

Când starea unui obiect eveniment cu auto-reset a fost semnalizată, ea rămâne semnalizată până când un singur fir de așteptare este eliberat; sistemul inițializează atunci automat starea pe nesemnalizat. Dacă nici un fir nu așteaptă, starea obiectului eveniment rămâne semnalizată. Procesele multiple pot avea identificatori la același obiect eveniment, activând utilizarea obiectului pentru sincronizarea interproces. Sunt disponibile următoarele mecanisme de partajare a obiectului:

- Un proces copil creat de funcția *CreateProcess* poate moșteni un identificator la un obiect eveniment, în cazul în care parametrul *lpEventAttributes* al funcției *CreateEvent* activează moștenirea.
- Un proces poate specifica identificatorul la obiectul eveniment printr-un apel la funcția *DuplicateHandle* pentru crearea unui identificator duplicat ce poate fi utilizat de către un alt proces.
- Un proces poate specifica denumirea obiectului eveniment printr-un apel la funcțiile *OpenEvent* sau *CreateEvent.*

Apelați funcția *CloseHandle* pentru închiderea identificatorului. Sistemul închide automat identificatorul când procesul este încheiat. Obiectul eveniment este distrus de sistemul de operare când ultimul său identificator a fost închis.

Pentru a înțelege mai bine prelucrările efectuate de program cu obiectele eveniment, să analizăm programul *Three_Options.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Pentru a vedea cel mai bine efectele sincronizării, lansați programul *Three_Options.cpp* de trei ori cu dispunerea instanțelor în mozaic pe desktop. În primele două instanțe, selectați opțiunea de meniu Read, iar în a treia instanță selectați opțiunea Write. Dacă cei care au optat pentru Read (citire) pot efectua execuții simultane, cei care au optat pentru Write (scriere) trebuie să aștepte terminarea operațiilor de citire pentru ca operațiile de scriere să poată fi executate. Dacă inversați ordinea și selectați mai întâi operațiile de scriere, operațiile de citire trebuie să aștepte până când operațiile de citire își efectuează prelucrarea. Fiecare proces își arată starea curentă în bara de stare a ferestrei. Partea operativă a programului *Three_Options* este prelucrată în cadrul funcției *WndProc.*

CE ESTE INTERFAȚA CU DISPOZITIVELE GRAFICE (GDI)? C/C++ 1414

Interfața cu dispozitivele grafice (GDI) reprezintă un set de funcții de bibliotecă prin care se furnizează aplicațiilor Windows o interfață independentă de dispozitivele monitor sau imprimantă. Interfața GDI reprezintă un nivel între aplicație și diferitele tipuri de hardware. Interfața GDI scutește programatorul de efortul de a aborda direct fiecare dispozitiv deoarece interfața este cea care rezolvă diferențele de hardware. O aplicație Windows bine proiectată va funcționa la fel pe toate cnentele hardware curente și pe orice hardware nou, lansat de producători pe viitor, datorită interfeței cu dispozitivele grafice.

Toate funcțiile de interfață cu dispozitive grafice din Win32 utilizează valori pe 32 de biți pentru coordonatele interfeței, deși în Windows 95 și Win32 sistemul de operare ignoră cuvântul cel mai semnificativ care rezultă într-o valoare a coordonatelor pe 16 biți. Numai Windows NT poate utiliza în întregime valorile pe 32 de biți.

1415 MOTIVELE UTILIZĂRII INTERFEȚEI CU DISPOZITIVELE GRAFICE



Așa cum vă așteptați, programele dumneavoastră vor utiliza interfața cu dispozitivele grafice pentru generarea unor ieșiri în loc de a utiliza texte sau ieșiri ne-grafice. Sunt nenumărate motive pentru utilizarea interfeței cu dispozitivele grafice pentru controlul conținutului ferestrelor, între care următoarele:

- Puteți aplica același context de dispozitiv la mai multe dispozitive.
- Puteți formata ieșirea în contextul de dispozitiv înaintea trimiterii ei la dispozitiv.
- Puteți controla grafica și alte informații vizuale din ferestre cu un context de dispozitiv.
- Puteți controla aspectul ferestrei (după derulare, dimensionare etc.) cu ajutorul contextului de dispozitiv.
- Programele dumneavoastră pot să trimită simplu ieșiri anterior plasate într-un context de dispozitiv al ferestrei, la imprimantă sau la alt dispozitiv.

Multe programe utilizate anterior în această carte au utilizat contextul de dispozitiv pentru menținerea informațiilor în interiorul ferestrei, în următoarele secțiuni vă veți concentra mai îndeaproape asupra utilizării și avantajelor contextelor de dispozitiv.

1416 CONTEXTELE DE DISPOZITIV



Instrumentul de bază utilizat de Windows pentru o oferi independența de dispozitiv a unei aplicații este *contextul de dispozitiv* sau DC. Contextul de dispozitiv este o structură internă utilizată de Windows pentru menținerea informațiilor despre un dispozitiv de ieșire. În loc să trimită ieșirea direct la hardware, o aplicație poate trimite ieșirea la un context de dispozitiv. Windows o trimite apoi la hardware în locul programului.

Un context de dispozitiv conține întotdeauna un creion pentru desenarea liniilor, o pensulă pentru pictarea suprafețelor, un font pentru afișarea caracterelor și o serie de alte valori pentru controlul comportamentului contextului de dispozitiv. Dacă aplicația solicită un font diferit, aplicația trebuie să selecteze fontul în contextul de dispozitiv înaintea afișării textului. Selectarea unui font nou nu schimbă textul existent din zona client a ferestrei.

Puteți vizualiza interfața oferită de un context de dispozitiv în reprezentarea din Figura 1416.

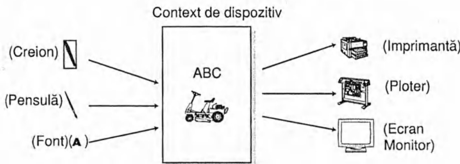


Figura 1416 *Modelul logic al unui context de dispozitiv.*

UTILIZAREA CONTEXTELOR DE DISPOZITIV PRIVATE

C/C++1417

În mod normal, aplicațiile partajează și regăsesc contextele de dispozitiv exact înaintea utilizării lor și le eliberează imediat după utilizare. Lucrul cu contextele de dispozitiv doar pentru scurte perioade de timp funcționează cel mai bine în cazul aplicațiilor care nu utilizează frecvent contexte de dispozitiv. O aplicație care cere utilizarea unui context pe o bază de sine stătătoare, poate crea o fereastră cu propriul context de dispozitiv privat, prin specificarea stilului de clasă *CD_OWNDC* în definiția clasei fereastră. Cu stilurile de clasă *CD_OWNDC* contextul de dispozitiv există pe durata întregii vieți a ferestrei. Aplicația va utiliza funcția *GetDC* pentru regăsirea unui identificator la contextul de dispozitiv, fără să necesite apelul la funcția *ReleaseDC* după încheierea prelucrării contextului respectiv. Când utilizați un context dispozitiv privat în programe care schimbă configurarea contextului de dispozitiv, prin producerea de modificări cum ar fi noi culori pentru text, creioane și pensule, acesta rămâne în funcțiune până când programul le modifică din nou.

ORIGINI ȘI EXTINDERI

C/C++1418

Un context de dispozitiv prezintă două moduri speciale de mapare, *MM_ISOTROPIC* și *MM_ANISOTROPIC*. Aceste două moduri de mapare folosesc două regiuni dreptunghiulare, *fereastra* și *portul de vizualizare (viewport)*, pentru derivarea factorului de redimensionare și o orientare. Fereastra este reprezentată în coordonate logice, iar portul de vizualizare în coordonate fizice. Împreună determină modul în care sistemul de operare mapează unitățile logice în unități fizice. Atât fereastra, cât și portul de vizualizare conțin o origine, o extindere x și una y. Originea este un punct care descrie oricare din cele patru colțuri. Originea portului de vizualizare este deplasamentul originii ferestrei. Extinderea x, este distanța pe orizontală de la origine la colțul opus. Extinderea y este distanța pe verticală de la origine la colțul opus.

Windows definește un factor de redimensionare pe orizontală prin împărțirea extinderii x a portului de vizualizare la extinderea x a ferestrei. Windows definește un factor de redimensionare pe verticală prin împărțirea extinderii y a portului de vizualizare la extinderea y a ferestrei. Factorii de redimensionare determină numărul de unități logice pe care Windows

le mapează la un număr de pixeli. În plus, față de determinarea factorilor de redimensionare, fereastra și portul de vizualizare determină orientarea unui obiect.

1419 OBȚINEREA UNUI CONTEXT DE DISPOZITIV PENTRU O FEREASTRĂ

C/C++

Programele dumneavoastră ar trebui să folosească contexte de dispozitiv pentru afișarea graficii și textelor atât pe un dispozitiv fereastră, cât și pe unul de imprimantă. Programele pot folosi fie funcția *GetDC*, fie funcția *GetDCEx* pentru regăsirea unui context de dispozitiv la o fereastră. Funcția *GetDC* regăsește identificatorul unui context de dispozitiv (DC) pentru fereastra specificată. Contextul de dispozitiv de afișare poate fi utilizat în funcții GDI ulterioare, pentru desenarea zonei client a ferestrei. Funcția *GetDCEx* este o extensie a funcției *GetDC* care oferă aplicației un control mai eficient asupra operațiunilor de decupare dintr-o zonă client. Veți implementa funcția *GetDCEx* potrivit prototipului de mai jos:

```
HDC GetDCEx(
   HWND hWnd,       // identificator pentru fereastra
   HRGN hrgnClip,   // identificator pentru regiunea de decupare
   DWORD flags      // indicatoare de creare a contextului de
                    // dispozitiv
);
```

Parametrul *hWnd* identifică fereastra în care se va desena. Parametrul *hrgnClip* specifică regiunea de decupare care poate fi combinată cu regiunea vizibilă a ferestrei client. Parametrul *flags* specifică modul în care este creat contextul de dispozitiv. Parametrul *flags* poate fi o combinație a valorilor înscrise în Tabelul 1419.

Valoare	**Semnificație**
DCX_WINDOW	Returnează un context de dispozitiv corespunzător unui dreptunghi fereastră și nu unui dreptunghi client.
DCX_CACHE	Returnează un context de dispozitiv din rezerva cache și nu fereastra *OWNDC* sau *CLASSDC*. Suprascrie valorile *CS_OWNDC* și *CS_CLASSDC*.
DCX_PARENTCLIP	Utilizează regiunea vizibilă a ferestrei părinte. Biții de stil *WS_CLIPCHILDREN* și *CS_PARENTDC* ai părintelui sunt ignorați. Originea contextului de dispozitiv este fixată în colțul din stânga sus a ferestrei identificate de *hWnd*.
DCX_CLIPSIBLINGS	Exclude regiunile vizibile ale tuturor ferestrelor copil de deasupra ferestrei identificate de *hWnd*.
DCX_CLIPCHILDREN	Exclude regiunile vizibile ale tuturor ferestrelor copil de sub fereastra identificată de *hWnd*.
DCX_NORESETATTRS	Nu inițializează atributele acestui context de dispozitiv la valorile implicite când contextul de dispozitiv este eliberat.

Valoare	Semnificație
DCX_LOCKWINDOWUPDATE	Permite desenarea în contextul de dispozitiv chiar dacă există un apel la *LockWindowUpdate* în execuție, care în caz contrar ar exclude această fereastră. Această valoare este folosită pentru a permite programelor să deseneze în timpul unei operații de căutare.
DCX_EXCLUDERGN	Regiunea de decupare identificată de *hrgnClip* este exclusă din regiunea vizibilă a contextului de dispozitiv returnat de *GetDCEx*.
DCX_INTERSECTRGN	Regiunea de decupare identificată de *hrgnClip* este intersectată cu regiunea vizibilă a contextului de dispozitiv returnat de *GetDCEx*.
DCX_VALIDATE	Dacă este specificat cu *DCX_INTERSECTUPDATE* determină completa validare a contextului de dispozitiv returnat de *GetDCEx*. Utilizarea acestei funcții cu *DCX_INTERSECTUPDATE* și *DCX_VALIDATE* este identică cu utilizarea funcției *BeginPaint*.

Tabelul 1419 *Valorile posibile ale parametrului **flags**.*

Când contextul de dispozitiv nu aparține unei clase ferestre, funcția *ReleaseDC* trebuie apelată pentru a elibera contextul de dispozitiv după desenare. Deoarece numai cinci contexte de dispozitiv obișnuite sunt disponibile în orice moment, eșuarea eliberării unui context de dispozitiv poate împiedica accesul altor aplicații la el. Atât *GetDC*, cât și *GetDCEx* returnează un context de dispozitiv care aparține clasei fereastră dacă programul specifică stilul *CS_CLASSDC*, *CS_OWNDC* sau *CS_PARENTDC* în structura WNDCLASS, în momentul înregistrării clasei.

Pentru a înțelege mai bine prelucrările efectuate de funcția *GetDCEx*, analizați programul *Draw_Hallow.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Draw_Hallow.cpp* utilizează funcția *GetDCEx* pentru obținerea unui context de dispozitiv pentru zona client a unei ferestre, excluzând o regiune. Apoi programul trasează un dreptunghi gri în zona client, excluzând regiunea internă (de culoare albă). Programul *Draw_Hallow.cpp* efectuează prelucrările operative în funcția *WndProc*.

CREAREA UNUI CONTEXT DE DISPOZITIV PENTRU O IMPRIMANTĂ

C/C++1420

Programele dumneavoastră trebuie să utilizeze contextele de dispozitiv pentru a desena ieșiri pentru diferite dispozitive. În timp ce Windows deține un context de dispozitiv încorporat (privat sau public), alte dispozitive de ieșire nu au contexte preexistente. Programele dumneavoastră trebuie să folosească funcția *CreateDC* pentru crearea unui context de dispozitiv pentru un dispozitiv cu denumire specificată. Veți implementa funcția *CreateDC* potrivit prototipului de mai jos:

```
HDC CreateDC(
    LPCTSTR lpszDriver,     // pointer la sirul cu numele
                            // driverului
    LPCTSTR lpszDevice,     // pointer la sirul cu numele
                            // dispozitivului
```

```
    LPCTSTR lpszOutput,  // nu il utilizati; dati-i valoarea
                         // NULL
    CONST DEVMODE *lpInitData // pointer la datele imprimantei
                              // optionale
);
```

Funcția *CreateDC* acceptă parametrii prezentați în Tabelul 1420.1.

Parametru	Descriere
lpszDriver	Pentru aplicațiile scrise în versiunile Windows anterioare, acest parametru specifică numele fișierului (fără extensie) al driverului de dispozitiv. În Windows 95 și în aplicațiile Win32, acest parametru este ignorat și trebuie să fie NULL, cu o excepție: puteți obține un context de dispozitiv de afișare prin specificarea șirului *DISPLAY* terminat cu NULL. Dacă acest parametru este *DISPLAY*, toți ceilalți parametri trebuie să fie NULL. Pentru Windows NT: *lpszDriver* indică un șir de caractere care specifică fie *DISPLAY* pentru un driver de afișare, fie numele unui driver de imprimantă, care este de obicei *WINSPOOL*.
lpszDevice	Indică un șir de caractere terminat cu NULL care specifică numele unui anumit dispozitiv de ieșire utilizat, prezentat în Print Manager (de exemplu „Epson FX-80"). Nu este numele modelului imprimantei, ci un nume intern sistemului Windows. Parametrul *lpszDevice* este obligatoriu.
lpszOutput	Acest parametru este ignorat de Windows. Nu îl utilizați în aplicațiile Win32. Aplicațiile bazate pe Win32 trebuie să stabilească acest parametru la NULL. El există pentru a furniza compatibilitate cu aplicațiile scrise în versiunile anterioare de Windows.
lpInitData	Indică o structură *DEVMODE* care conține datele de inițializare, specifice dispozitivului, pentru driverul de dispozitiv. Funcția *DocumentProperties* regăsește această structură și o completează pentru un dispozitiv specificat. Parametrul *lpInitData* trebuie să fie NULL, dacă driverul dispozitivului utilizează inițializarea implicită (dacă există) specificată de utilizator.

***Tabelul 1420.1** Parametrii funcției **CreateDC**.*

Dacă funcția reușește, valoarea returnată este identificatorul la un context de dispozitiv pentru dispozitivul specificat. Dacă funcția eșuează, valoarea returnată va fi NULL.

Așa cum observați și în Tabelul 1420.1, funcția *CreateContext* așteaptă ca ultim parametru un pointer la o structură *DEVMODE*, care conține datele de inițializare specifice ale driverului dispozitivului. Interfața Win32 API definește structura *DEVMODE* în felul următor:

```
typedef struct _devicemode {
    TCHAR dmDeviceName[32];
    WORD dmSpecVersion;
    WORD dmDriverVersion;
    WORD dmSize;
    WORD dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
```

```
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
    short dmCollate;
    TCHAR dmFormName[32];
    WORD dmUnusedPadding;
    USHORT dmBitsPerPel;
    DWORD dmPelsWidth;
    DWORD dmPelsHeight;
    DWORD dmDisplayFlags;
    DWORD dmDisplayFrequency;
} DEVMODE;
```

Structura de date *DEVMODE* conține informații despre instalarea și mediul dispozitivului de imprimare. Tabelul 1420.2 descrie în detaliu membrii structurii *DEVMODE*.

Membru	Descriere
dDmDeviceName	Specifică numele imprimantei acceptat de driver; de exemplu „PLC/HP LaserJet", unde numele comercial este *PCL/HP LaserJet*. Acest șir este unic între driverele de dispozitiv.
dmSpecVersion	Specifică numărul versiunii specificației datelor de inițializare pe care se bazează structura contextului de dispozitiv.
dmDriverVersion	Specifică numărul versiunii de driver al imprimantei atribuit de constructorul driverului.
dmSize	Specifică dimensiunea, în octeți, a structurii DEVMODE, fără includerea datelor private specifice driverului (*dmDriverDate*) care pot urma membrilor publici ai structurii. Aplicațiile pot folosi acest membru pentru obținerea numărului de octeți ai datelor publice, indiferent de versiunea structurii *DEVMODE* utilizate.
dmDriverExtra	Conține numărul de octeți ai datelor private de driver care urmează acestei structuri. Dacă un driver de dispozitiv nu utilizează informațiile specifice dispozitivului, stabiliți acest membru la zero.
dmFields	Precizează care din membrii structurii *DEVMODE* rămași au fost inițializați. Bitul zero (definit ca *DM_ORIENTATION*) corespunde lui *dmOrientation;* bitul 1 (definit ca *DM_PAPERSIZE*) corespunde lui *dmPaperSize* și așa mai departe. Un driver de imprimantă acceptă numai acei membri ai structurii *DEVMODE* care corespund tehnologiei imprimantei.
dmOrientation	Selectează orientarea hârtiei. Acest membru poate fi fie *DMORIENT_PORTRAIT* (1), fie *DMORIENT_LANDSCAPE* (2).

(continuare)

Membru	Descriere
dmPaperSize	Selectează dimensiunea hârtiei pe care se tipărește. Acest membru poate primi valoarea zero dacă lungimea și lățimea hârtiei sunt ambele date de membrii *dmPaperLength* și *dmPaperWidth*. În caz contrar, membrul *dmPaperSize* poate primi una din valorile predefinite cuprinse în Tabelul 1420.3.
dmPaperLength	Suprascrie lungimea hârtiei specificate de membrul *dmPaperSize*, fie cu dimensiunile definite de utilizator, fie cu dispozitive cum ar fi imprimantele matriceale care imprimă pe o pagină de lungime arbitrară. Această valoare, împreună cu toate celelalte valori din structură care specifică o lungime fizică, este precizată în zecimi de milimetru.
dmPaperWidth	Suprascrie lățimea hârtiei specificate de membrul *dmPaperSize*.
dmScale	Specifică factorul după care va fi scalată ieșirea la imprimantă. Mărimea paginii aparente este scalată, pornind de la dimensiunea fizică a paginii, cu un factor de *dmScale/100*. De exemplu, o pagină tip *letter* (8,5/11 inci) cu o valoare *dmScale* de 50 va conține la fel de multe date ca o pagină de 17/22 inci pentru că textul și grafica de ieșire vor fi dimensionate la jumătatea valorii lor originare.
dmCopies	Selectează numărul de copii imprimate dacă dispozitivul acceptă copii multiple de pagini.
dmDefaultSource	Rezervat – trebuie să fie zero.
dmPrintQuality	Specifică rezoluția imprimantei. Există patru valori independente de dispozitiv, predefinite: *DMRES_HIGH* *DMRES_LOW* *DMRES_MEDIUM* *DMRES_DRAFT* Dacă este dată o valoare pozitivă ne-constantă, ea specifică numărul de puncte pe inci (DPI), fiind deci dependentă de dispozitiv.
dmColor	La imprimantele color, comută de la color la monocrom. Valorile posibile pentru *dmColor* sunt următoarele: *DMCOLOR_COLOR* *DMCOLOR_MONOCHROME*
dmDuplex	Selectează imprimarea tip duplex sau pe ambele părți la imprimantele cu posibilitatea de imprimare duplex. Valorile posibile pentru *dmDuplex* sunt următoarele: *DMDUP_SIMPLEX* *DMDUP_HORIZONTAL* *DMDUP_VERTICAL*
dmYResolution	Specifică rezoluția y a imprimantei, în puncte pe inci. Dacă imprimanta inițializează acest membru, atunci membrul *dmPrintQuality* specifică rezoluția x a imprimantei, în puncte pe inci.

Membru	**Descriere**	
dmTTOption	Specifică modul de imprimare a fonturilor *TrueType*®. Acest membru poate lua una din următoarele valori:	
	DMTT_BITMAP	Imprimă fonturi *TrueType* ca grafică. Aceasta este o acțiune implicită a imprimantelor matriceale.
	DMTT_DOWNLOAD	Încarcă în memorie fonturile *TrueType* ca fonturi soft. Aceasta este acțiunea implicită a imprimantelor *Hewlett-Packard* care utilizează *Printer Control Language* (PCL).
	DMTT_SUBDEV	Substituie fonturile dispozitivului cu fonturi *TrueType*. Aceasta este acțiunea implicită a imprimantelor *PostScript*®.
dmCollate	Specifică dacă se va folosi juxtapunerea la imprimarea copiilor multiple. Utilizarea lui *DMCOLLATTE_FALSE* oferă o imprimare mai rapidă și mai eficientă deoarece datele sunt trimise la driverul dispozitivului numai o dată, indiferent de numărul de copii cerute. Se comunică imprimantei să tipărească pagina din nou. (Acest membru este ignorat, cu excepția cazului în care driverul de imprimantă indică suport pentru juxtapunere prin stabilirea membrului *dmFields* la *DM_COLLATE*). Acest membru poate lua una din următoarele valori:	
DMCOLLATE_TRUE	Juxtapunere la copii multiple	
DMCOLLATE_FALSE	Fără juxtapunere la copii multiple	
dmFormName	Numai pentru Windows NT: specifică numele formularului pentru utilizare; de exemplu, „Letter" sau „Legal". Setul complet al numelor poate fi regăsit cu funcția *EnumForms*.	
dmUnusedPadding	Aliniază o structură la o limitare *DWORD*. Rezervat pentru versiunile următoare de Windows. Nu se recomandă utilizarea sau referirea acestei valori.	
dmBitsPerPel	Arată culoarea rezoluției unui dispozitiv de afișare, în biți pe pixeli (de exemplu: 4 biți pentru 16 culori, 8 biți pentru 256 de culori sau 16 biți pentru 65536 de culori).	
dmPelsWidth	Specifică lățimea, în pixeli, a suprafeței vizibile a dispozitivului.	
dmPelsHeight	Specifică înălțimea, în pixeli, a suprafeței vizibile a dispozitivului.	
dmDisplayFlags	Specifică modul dispozitivului de afișare. Acest membru poate lua una din următoarele valori:	
DM_GRAYSCALE	Precizează faptul că dispozitivul de afișare nu este color. Dacă acest indicator nu este fixat, se presupune că dispozitivul este color.	
DM_INTERLACED	Precizează faptul că modul de afișare este întrețesut. Dacă acest indicator nu este fixat, se presupune ca valabil modul neîntrețesut.	
dmDisplayFrequency	Specifică frecvența, în herți (oscilații pe secundă), unui dispozitiv de afișare într-un anumit mod.	

***Tabelul 1420.2** Membrii structurii **DEVMODE**.*

Așa cum descrie Tabelul 1420.2, parametrul *dmPaperSize* acceptă constante pre-definite care corespund celor mai cunoscute dimensiuni de pagină pe plan internațional. Tabelul 1420.3 enumeră o parte din aceste constante.

Valoare	Semnificație
DMPAPER_LETTER	Letter, 8 1/2 pe 11 inci
DMPAPER_LEGAL	Legal, 8 1/2 pe 14 inci
DMPAPER_A4	A4, 210 pe 297 milimetri
DMPAPER_LEDGER	Ledger, 17 pe 11 inci
DMPAPER_STATEMENT	Statement, 5 1/2 pe 8 1/2 inci
DMPAPER_EXECUTIVE	Executive, 7 1/4 pe 10 1/2 inci
DMPAPER_FOLIO	Folio, 8 1/2 pe 13 inci
DMPAPER_QUARTO	Quarto, 215 pe 275 milimetri
DMPAPER_11X17	11 pe 17 inci
DMPAPER_ENV_10	#10, 4 1/8 pe 9 ½ inci, plic
DMPAPER_FANFOLD_US	US Std, 14 7/8 pe 11 inci, hârtie pentru scrisoare
DMPAPER_FANFOLD_LGL_GERMAN	German Legal, 8 1/2 per13 inci, hârtie pentru scrisoare

Tabelul 1420.3 *Câteva din valorile pentu dimensiunea hârtiei.*

Datele private ale unui driver de dispozitiv urmează membrului *dmDisplayMode*. Membrul *dmDriverExtra* specifică numărul de octeți de date private. Aplicațiile scrise în versiunile de Windows mai vechi utilizează parametrul *lpszOutput* pentru specificarea unui nume de port sau imprimarea în fișier. Aplicațiile bazate pe Win32 nu necesită specificarea unui nume de port. Aplicațiile bazate pe Win32 pot imprima într-un fișier prin apelarea funcției *StartDoc* cu o structură *DOCINFO* al cărei membru *lpszOutput* conține calea de acces a fișierului de ieșire. Când nu mai aveți nevoie de contextul de dispozitiv, apelați funcția *DeleteDC* pentru a-l șterge.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateDC*, analizați programul *Print_File.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Print_File* tipărește o singură linie de text la imprimantă când utilizatorul selectează articolul de meniu *Test!*. Funcția *CreateDC* creează contextul de dispozitiv pentru imprimantă. CD-ROM-ul citește numele imprimantei ca „HP LaserJet 4 Plus", dar puteți schimba numele cu cel corespunzător driverului sistemului dumneavoastră sau apelați funcția *EnumPrinters* pentru a afla ce imprimantă aveți conectată la calculator. Funcția *WndProc* conține prelucrarea operativă a programului *Print_File.cpp*.

1421 UTILIZAREA LUI CREATECOMPATIBLEDC PENTRU CREAREA UNUI CONTEXT DE DISPOZITIV DE MEMORIE



În secțiunile precedente ați în0vățat cum se pot utiliza contextele de dispozitiv pentru a genera ieșiri la monitor sau la imprimantă. Totuși, programele dumneavoastră nu pot efectua operații de desenare direct pe un context de dispozitiv. Funcția *CreateCompatibleDC* creează un context de dispozitiv de memorie (DC) compatibil cu dispozitivul specificat. Înainte ca o

aplicație să poată utiliza un context de dispozitiv de memorie pentru operații de desenare, trebuie să selectați un bitmap cu înălțimea și lungimea corecte în contextul de dispozitiv. O dată selectat acest bitmap, contextul de dispozitiv poate fi utilizat la elaborarea imaginilor care vor fi copiate pe ecran sau la imprimantă. De fiecare dată când programul lucrează cu formate bitmap (despre care vom vorbi ulterior), programele vor plasa acel bitmap mai întâi într-un context de dispozitiv de memorie, apoi îl vor copia în contextul de dispozitiv specificat. Programele vor utiliza funcția *CreateCompatibleDC* după următorul prototip:

```
HDC CreateCompatibleDC(HDC hdc)
    // identificator pentru contextul de dispozitiv de memorie
);
```

Parametrul *hdc* identifică un context de dispozitiv. Dacă este NULL, funcția *CreateCompatibleDC* produce un context de dispozitiv de memorie compatibil cu ecranul aplicației curente. Dacă *CreateCompatibleDC* reușește, valoarea returnată este un identificator pentru un context de dispozitiv de memorie. Dacă eșuează, valoarea returnată este NULL.

Funcția *CreateCompatibleDC* poate fi utilizată cu dispozitive care acceptă operațiile de rastru. Aplicația poate determina dacă un dispozitiv acceptă aceste operații prin apelarea funcției *GetDeviceCaps*. Când nu mai aveți nevoie de contextul de dispozitiv de memorie, apelați *DeleteDC* pentru a-l șterge.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateCompatibleDC*, analizați programul *Draw_Bitmap.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Draw_Bitmap.cpp* încarcă un bitmap și îl plasează într-un context de dispozitiv de memorie, apoi îl copiază în zona client a ferestrei. Ca de obicei, codul operativ al programului este prelucrat în funcția *WndProc*.

PERICOLELE FUNCȚIEI CREATEDC C/C++1422

Așa cum ați învățat în secțiunea 1420, programele dumneavoastră vor utiliza frecvent funcția *CreateDC* pentru a obține un context de dispozitiv al unei imprimante. Ele pot însă să utilizeze funcția și pentru obținerea contextului de dispozitiv al ecranului (monitorul hardware, nu o singură zonă client a unei ferestre sau chiar a zonei client din desktop). Când utilizați *CreateDC* pentru a obține contextul de dispozitiv pentru ecran, programul poate efectiv desena oriunde pe ecran, nu numai în limitele unei zone a programului. În plus față de obținerea unor rezultate potențial imprevizibile, acest procedeu nu este consistent cu standardul Windows. Când încearcă să obțină un context de dispozitiv la o fereastră pe ecran, programele dumneavoastră trebuie întotdeauna să folosească *GetDC* sau *BeginPaint*, și nu funcția *CreateDC*.

UTILIZAREA FUNCȚIEI CREATEFONT C/C++1423

Pe măsură ce lucrați mai îndeaproape cu contexte de dispozitiv și afișați multe lucruri interesante pe ecran, veți simți nevoia să modificați fontul afișat într-o fereastră sau să creați un font personalizat, ceea ce programele pot efectua cu funcțiile *CreateFont* sau *CreateFontIndirect*. Dacă manipulați fonturi diferite, utilizați funcția *CreateFontIndirect*, iar dacă manipulați un singur font, utilizați funcția *CreateFont*. Funcția *CreateFont* produce un font logic (o definiție numerică de font) cu caracteristici specifice. Ulterior puteți selecta

fontul logic ca font pentru orice dispozitiv. Programele vor utiliza funcția *CreateFont* potrivit prototipului de mai jos:

```
HFONT CreateFont(
    int nHeight,                  // inaltimea logica a fontului
    int nWidth,                   // latime logica medie a
                                  // caracterului
    int nEscapement,              // unghiul de scapare
    int nOrientation,             // unghi de orientare a liniei de
                                  // baza
    int fnWeight,                 // grosime font
    DWORD fdwItalic,              // indicator de atribut italic
    DWORD fdwUnderline,           // indicator de atribut subliniere
    DWORD fdwStrikeOut,           // indicator de atribut taiere
    DWORD fdwCharSet,             // identificator set de caractere
    DWORD fdwOutputPrecision,     // precizie iesire
    DWORD fdwClipPrecision,       // precizie decupare
    DWORD fdwQuality,             // calitate iesire
    DWORD fdwPitchAndFamily,      // densitate si familie de
                                  // caractere
    LPCTSTR lpszFace              // pointer la numele de tip de
                                  // caracter
);
```

Funcția *CreateFont* acceptă parametrii descriși în Tabelul 1423.1.

Parametru	Descriere
nHeight	Specifică înălțimea, în unități logice, a caracterului sau celulei caracterului unui font. Valoarea înălțimii caracterului (cunoscută și ca înălțime *em*) este valoarea înălțimii celulei caracterului minus valoarea de interliniere (spațiul dintre rânduri). Sistemul de mapare a fonturilor interpretează valoarea specificată în *nHeight*. Dacă valoarea lui *nHeight* este mai mare decât zero, sistemul de mapare transformă această valoare în unități de dispozitiv și o compară cu înălțimea celulei fonturilor disponibile. Dacă valoarea lui *nHeight* este egală cu zero, sistemul de mapare a fontului utilizează valoarea înălțimii implicite în comparații. Dacă valoarea lui *nHeight* este mai mică decât zero, sistemul de mapare a fontului transformă această valoare în unități de dispozitiv și compară valoarea sa absolută cu înălțimea caracterului fonturilor disponibile. Pentru toate operațiile de comparare a înălțimilor, sistemul de mapare a fontului caută fontul cel mai mare care nu depășește dimensiunea cerută. Această mapare intervine când fontul este utilizat pentru prima oară.
nWidth	Specifică lățimea medie, în unități logice, a caracterelor respectivului font. Dacă valoarea este zero, sistemul de mapare a fontului caută cea mai apropiată valoare de „potrivire". Această valoare de „potrivire" se determină prin compararea valorilor absolute ale diferenței dintre raportul de aspect al dispozitivului curent și raportul digitalizat de aspect al fonturilor disponibile.

Parametru	Descriere
nEscapement	Specifică unghiul, în zecimi de grad, între vectorul de *escapement* (depășire) și axa x a dispozitivului. Vectorul de escapement este paralel cu linia de bază a unui rând de text. Sub Windows NT, când modul grafic este fixat pe *GM_ADVANCED*, puteți specifica unghiul de escapement al șirului independent de orientarea unghiului caracterelor șirului. Când modul grafic este fixat pe *GM_COMPATIBLE*, parametrul *nEscapement* specifică atât vectorul de escapement, cât și unghiul de orientare. În general, se fixează *nEscapement* și *nOrientation* la aceeași valoare. Sub Windows 95, parametrul *nEscapement* specifică atât vectorul de escapement, cât și unghiul de orientare. Trebuie să fixați, ca și în Windows NT, *nEscapement* și *nOrientation* la aceeași valoare.
nOrientation	Specifică unghiul, în zecimi de grad, între linia de bază a fiecărui caracter și axa x a dispozitivului.
fnWeight	Specifică grosimea fontului în intervalul 0 la 1000. De exemplu, 400 este normal, iar 700 este îngroșat. Dacă valoarea este zero, se utilizează valoarea implicită. Parametrul *fnWeight* poate lua una din valorile prezentate în Tabelul 1423.2.
fdwItalic	Dacă e TRUE, specifică un font italic.
fdwUnderline	Dacă e TRUE, specifică un font subliniat.
fdwStrikeOut	Dacă e TRUE, specifică un font tăiat.
fdwCharSet	Specifică setul de caractere. Valorile predefinite sunt prezentate în Tabelul 1423.3. Valoarea *OEM_CHARSET* specifică setul de caractere dependent de sistemul de operare. Puteți să utilizați *DEFAULT_CHARSET* ca numele și dimensiunea unui font să descrie complet fontul logic. Dacă numele fontului specificat nu există, un font din oricare set de caractere poate substitui respectivul font. Trebuie deci, să utilizați destul de rar *DEFAULT_CHARSET* pentru evitarea rezultatelor neașteptate. În sistemul de operare pot exista fonturi cu alte seturi de caractere. Dacă o aplicație utilizează un font cu un set de caractere necunoscut, ea nu va trebui să convertească sau să interpreteze șirurile redate cu acest font. Parametrul *fdwCharSet* este important pentru procesul de mapare a fonturilor. Pentru a avea rezultate consistente, precizați un anumit set de caractere. Dacă specificați un nume de tip în parametrul *lpszFace*, asigurați-vă că valoarea *fdwCharSet* corespunde setului de caractere al tipului specificat în *lpszFace*.
fdwOutputPrecision	Specifică precizia la ieșire. Precizia la ieșire definește precizia modului în care ieșirea corespunde cu înălțimea, lățimea, orientarea, escapement, densitatea și tipul fontului. Parametrul poate lua una din valorile enumerate în Tabelul 1423.4. Aplicațiile pot utiliza valorile *OUT_DEVICE_PRECIS*, *OUT_RASTER_PRECIS* și *OUT_TT_PRECIS* pentru controlul modului în care sistemul de mapare al fontului alege un font în condițiile în care sistemul de operare conține mai multe fonturi cu denumire specificată. De exemplu, dacă sistemul de operare conține fontul denumit Symbol în rastru și forma TrueType, specificarea *OUT_TT_PRECIS* forțează sistemul de mapare al fontului să aleagă versiunea TrueType. Specificarea valorii *OUT_TT_ONLY_PRECIS* forțează sistemul de mapare al fontului să aleagă un font TrueType, chiar dacă trebuie să substituie un font TrueType cu altă denumire.

(continuare)

Parametru	Descriere
fdwClip Precision	Specifică precizia de decupare. Precizia de decupare definește modul de decupare al caracterelor care sunt parțial în afara regiunii de decupare. Parametrul poate lua una din valorile prezentate în Tabelul 1423.5.
fdwQuality	Conține calitatea la ieșire. Calitatea la ieșire definește precizia cu care interfața de dispozitiv grafic trebuie să încerce să stabilească echivalența atributelor fontului logic cu cele ale fontului fizic real. Parametrul poate lua una din valorile înscrise în Tabelul 1423.6.
fdwPitchAnd Family	Specifică densitatea și familia fontului. Cei doi biți mai puțin semnificativi specifică densitatea fontului și poate lua una din următoarele valori: *FIXED_PITCH* *DEFAULT_PITCH* *VARIABLE_PITCH* Cei 4 biți mai semnificativi specifică familia fontului și poate lua una din valorile înscrise în Tabelul 1423.7. O aplicație poate specifica o valoare pentru parametrul *fdwPitchAndFamily* utilizând operatorul boolean *OR* pentru a uni o constantă de densitate cu una de familie. Familiile de fonturi descriu aspectul general al fontului. Scopul lor este specificarea fontului când tipul exact cerut nu este disponibil.
lpszFace	Indică un șir terminat cu NULL care specifică numele tipului de font. Lungimea acestui șir trebuie să nu depășească 32 de caractere, inclusiv terminatorul de șir. Funcția *EnumFontFamilies* poate fi utilizată pentru enumerarea numelor de tipuri aparținând tuturor fonturilor curente disponibile. Dacă *lpszFace* este NULL sau indică un șir vid, interfața de dispozitive grafice utilizează primul font care corespunde celorlalte atribute specificate.

Tabelul 1423.1 *Parametrii funcției* ***CreateFont***.

Așa cum observați în Tabelul 1423.1, programele dumneavoastră pot fixa o varietate de grosimi predefinite de fonturi la crearea unui font logic. Tabelul 1423.2 enumeră valorile posibile ale parametrului *nWeight:*

Valoare	Grosime	Valoare	Grosime
FW_DONTCARE	0	*FW_SEMIBOLD*	600
FW_THIN	100	*FW_BOLD*	700
FW_EXTRALIGHT	200	*FW_EXTRABOLD*	800
FW_LIGHT	300	*FW_ULTRABOLD*	800
FW_NORMAL	400	*FW_HEAVY*	900
FW_REGULAR	400	*FW_BLACK*	900
FW_MEDIUM	500		

Tabelul 1423.2 *Valorile posibile ale parametrului* ***nWeight***.

Așa cum observați în Tabelul 1423.1, funcția *CreateFont* permite specificarea unui set de caractere predefinite pentru un anumit font. Tabelul 1423.3 enumeră valorile de constante posibile ale setului de caractere.

Set de caractere predefinit		
ANSI_CHARSET	*DEFAULT_CHARSET*	*SYMBOL_CHARSET*
SHIFTJIS_CHARSET	*GB2312_CHARSET*	*HANGEUL_CHARSET*
CHINESEBIG5_CHARSET	*OEM_CHARSET*	
Numai în Windows 95:		
JOHAB_CHARSET	*HEBREW_CHARSET*	*ARABIC_CHARSET*
GREEK_CHARSET	*TURKISH_CHARSET*	*THAI_CHARSET*
EASTEUROPE_CHARSET	*RUSSIAN_CHARSET*	
MAC_CHARSET	*BALTIC_CHARSET*	

Tabelul 1423.3 *Constantele predefinite ale setului de caractere.*

Precizia la ieșire definește precizia corespondenței ieșirii cu înălțimea, lățimea, orientarea și alte caracteristici ale fontului. Pentru controlul preciziei la ieșire, parametrul *fdwOutputPrecision* poate lua una din valorile prezentate în Tabelul 1423.4.

Valoare	**Semnificație**
OUT_CHARACTER_PRECIS	Neutilizat
OUT_DEFAULT_PRECIS	Conține modul de lucru implicit al sistemului de mapare al fontului.
OUT_DEVICE_PRECIS	Cere sistemului de mapare al fontului să aleagă un font de dispozitiv când sistemul conține mai multe fonturi cu același nume.
OUT_OUTLINE_PRECIS	Sub Windows NT, valoarea cere sistemului de mapare al fontului să aleagă din fonturile *TrueType* și alte fonturi de contur. Sub Windows 95, această valoare nu este utilizată.
OUT_RASTER_PRECIS	Cere sistemului de mapare al fontului să aleagă un font de rastru când sistemul conține mai multe fonturi cu același nume.
OUT_STRING_PRECIS	Această valoare nu este utilizată de sistemul de mapare al fontului, dar este returnată când sunt enumerate fonturile de rastru.
OUT_STROKE_PRECIS	Sub Windows NT această valoare nu este utilizată de sistemul de mapare al fontului, dar este returnată de funcție când sunt enumerate alte fonturi TrueType, vectoriale și de contur. Sub Windows 95 această valoare este utilizată pentru maparea fonturilor vectoriale și este returnată când sunt enumerate fonturi TrueType sau vectoriale.
OUT_TT_ONLY_PRECIS	Cere sistemului de mapare al fontului să aleagă numai din fonturile TrueType. Dacă în sistem nu sunt instalate fonturi TrueType, sistemul de mapare al fontului returnează modul de lucru implicit.
OUT_TT_PRECIS	Cere sistemului de mapare al fontului să aleagă un font TrueType când sistemul conține mai multe fonturi cu același nume.

Tabelul 1423.4 *Valorile posibile ale parametrului* ***fdwOutputPrecision***.

În același mod în care programele dumneavoastră pot controla precizia desenării fonturilor în contextul de dispozitiv, la fel pot controla gradul de precizie în care Windows decupează caracterele aflate parțial în afara regiunii de decupare. Tabelul 1423.5 descrie valorile posibile ale parametrului *fdwClipPrecision*.

Valoare	Precizie
CLIP_DEFAULT_PRECIS	Specifică modul implicit de decupare.
CLIP_CHARACTER_PRECIS	Neutilizată.
CLIP_STROKE_PRECIS	Neutilizat de sistemul de mapare al fontului, dar returnat când sunt enumerate fonturile de rastru, fontele vectoriale sau cele TrueType. Pentru compatibilitate, în Windows NT, această valoare este întotdeauna returnată când se enumeră fonturi.
CLIP_MASK	Neutilizată.
CLIP_EMBEDDED	Aplicațiile trebuie să specifice acest indicator pentru utilizarea unui font încorporat protejat la scriere.
CLIP_LH_ANGLES	Când este utilizată această valoare, rotația tuturor fonturilor depinde de orientarea la dreapta sau la stânga a sistemului de coordonate. Dacă nu este utilizat, fonturile de dispozitiv se rotesc întotdeauna în sensul invers al acelor de ceasornic, dar rotația altor fonturi este dependentă de orientarea sistemului de coordonate.
CLIP_TT_ALWAYS	Neutilizată.

***Tabelul 1423.5** Valorile posibile ale parametrului **fdwClipPrecision**.*

În plus față de precizia la ieșire, programele dumneavoastră trebuie să controleze calitatea la ieșire. Există trei valori posibile ale calității la ieșire definite în parametrul *fdwQuality* descrise în Tabelul 1423.6.

Valoare	Semnificație
DEFAULT_QUALITY	Aspectul fontului nu are importanță.
DRAFT_QUALITY	Aspectul fontului este mai puțin important decât când este utilizată valoarea *PROOF_QUALITY*. Pentru fonturile de rastru ale interfeței cu dispozitivele grafice este activată dimensionarea, ceea ce înseamnă că sunt disponibile mai multe dimensiuni de fonturi, dar calitatea lor este mai redusă. Dacă este necesar sunt sintetizate fonturile îngroșate, înclinate, subliniate și tăiate.
PROOF_QUALITY	Calitatea caracterelor fontului este mai importantă decât corespondența exactă a atributelor fontului logic. Pentru fonturile de rastru ale interfeței cu dispozitivul grafic este dezactivată dimensionarea și este ales fontul cu cea mai apropiată mărime. Deși dimensiunea fontului ales nu poate fi mapată exact când este folosit *PROOF_QUALITY*, calitatea fontului este ridicată și nu există distorsiuni în aspect. Dacă este necesar, sistemul de operare sintetizează fonturile îngroșate, înclinate, subliniate și tăiate.

***Tabelul 1423.6** Valori posibile ale parametrului **fdwQuality**.*

În sfârșit, programele dumneavoastră pot specifica densitatea și familia fontului, ceea ce permite funcției *CreateFont* să realizeze o corespondență apropiată a fontului, dacă nu una

exactă. Tabelul 1423.7 descrie familiile de fonturi ce pot fi utilizate în parametrul *fdwPitchAndFamily*.

Valoare	Descriere
FF_DECORATIVE	Fonturi originale. De exemplu, Old English.
FF_DONTCARE	Programul nu ține cont de familie sau nu o știe.
FF_MODERN	Fonturi cu lățime constantă, cu sau fără serife. Exemplu: Pica, Elite și Courier New®.
FF_ROMAN	Fonturi cu lățime variabilă și cu serife. Exemplu: MS® Serif.
FF_SCRIPT	Fonturile sunt modelate să semene cu scrisul de mână. Exemplu: Script și Cursive.
FF_SWISS	Fonturi cu lățime variabilă și fără serife. Exemplu: MS Sans Serif.

Tabelul 1423.7 *Familiile de fonturi ce pot fi utilizate în parametrul* ***fdwPitchAndFamily***.

Dacă funcția *CreateFont* reușește, valoarea returnată este identificatorul fontului logic. Dacă funcția eșuează, valoarea returnată va fi NULL. Dacă nu mai aveți nevoie de font, apelați funcția *DeleteObject* pentru a-l șterge.

Pentru protejarea drepturilor de copyright ale producătorilor de fonturi pentru sistemele de operare Windows, aplicațiile trebuie să raporteze exact numele fontului selectat. Deoarece fonturile disponibile pot varia de la un sistem la altul, să nu considerați că fontul selectat este întotdeauna același cu fontul cerut. De exemplu, dacă cereți fontul cu numele „Palatino", dar nici un astfel de font nu e disponibil în sistem, sistemul de mapare al fontului îl va substitui cu un font asemănător, dar cu nume diferit. Întotdeauna menționați utilizatorului numele fontului selectat.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateFont*, analizați programul *Create_BigRoman.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Create_BigRoman* creează un font Times New Roman cu dimensiunea de 24/16. Programul utilizează apoi noul font creat pentru a tipări texte în zona client a ferestrei. Codul operativ al programului *Create_BigRoman* se află în funcția *WndProc*.

UTILIZAREA FUNCȚIEI ENUMFONTFAMILIES — C/C++1424

Așa cum ați învățat în secțiunea 1423, Windows grupează fonturile în familii pe baza unor caracteristici comune fiecărui font din cadrul familiei. Programele dumneavoastră pot utiliza funcția *EnumFontFamilies* pentru a enumera într-o familie de fonturi, fonturile disponibile pe dispozitivul specificat. Funcția *EnumFontFamilies* va fi în general utilizată pentru a obține fonturile care pot fi utilizate pe un anumit dispozitiv și pentru a obține un pointer la o structură *LOGFONT*. Această structură poate fi folosită de program împreună cu funcția *CreateFontIndirect* pentru crearea fontului pe dispozitivul specificat. Programele vor implementa funcția *EnumFontFamilies* potrivit prototipului de mai jos:

```
int EnumFontFamilies(
    HDC hdc,            // identificator pentru contextul de
                        // dispozitiv
    LPCTSTR lpszFamily, // pointer la un sir cu numele familiei
    FONTENUMPROC lpEnumFontFamProc, // pointer la o functie
```

```
                                         // callback
    LPARAM lParam    // adresa cu datele furnizate de aplicatie
);
```

Tabelul 1424.1 descrie parametrii acceptaţi de funcţia *EnumFontFamilies*.

Parametru	Descriere
hdc	Identifică contextul de dispozitiv.
lpszFamily	Indică un şir terminat cu NULL care specifică numele de familie al fonturilor dorite. Dacă *lpszFamily* este NULL, funcţia *EnumFontFamilies* selectează şi enumeră la întâmplare un font din fiecare familie de tipuri disponibilă.
lpEnumFontFamProc	Specifică adresa instanţei procedurii pentru funcţia callback definită de aplicaţie. Pentru informaţii privind funcţia callback, vezi funcţia *EnumFontFamProc*.
lParam	Indică datele furnizate de aplicaţie. Aceste date sunt transmise funcţiei callback împreună cu informaţiile despre font.

***Tabelul 1424.1** Parametrii funcţiei **EnumFontFamilies**.*

Dacă funcţia *EnumFontFamilies* reuşeşte, valoarea returnată este ultima valoare returnată de funcţia callback. Semnificaţia sa este dependentă de implementare. Funcţia *EnumFontFamilies* regăseşte numele stilurilor asociate cu fontul *TrueType*. Cu funcţia *EnumFontFamilies* pot fi regăsite informaţii despre stiluri neobişnuite de fonturi (de exemplu, Outline) care nu pot fi enumerate folosind funcţia *EnumFonts*. Pentru fiecare font cu numele de tip specificat de parametrul *lpszFamily*, funcţia *EnumFontFamilies* regăseşte informaţiile despre acel font şi le transmite funcţiei indicate de parametrul *lpEnumFontFamProc*. Funcţia callback definită de aplicaţie poate prelucra informaţiile despre font după cum se doreşte. Enumerarea continuă până când nu mai există alte fonturi sau funcţia callback returnează zero. Veţi utiliza funcţia callback în felul următor:

```
int CALLBACK EnumFontFamProc(
     ENUMLOGFONT * lpelf, // pointer la o structura ENUMLOGFONT
     NEWTEXTMETRIC * lpntm, // pointer la o structura
                            // NEWTEXTMETRIC
     int nFontType,         // tip font
     LPARAM lParam          // date definite de aplicatie
);
```

Aşa cum observaţi, funcţia callback acceptă patru parametri. Primul parametru este un pointer la o structură *ENUMLOGFONT*. Structura *ENUMLOGFONT* defineşte atributele fontului, numele complet al fontului şi stilul fontului. Interfaţa Win32 API defineşte structura *ENUMLOGFONT* în modul prezentat mai jos:

```
typedef struct tagENUMLOGFONT {
    LOGFONT elfLogFont;
    BCHAR   elfFullName[LF_FULLFACESIZE];
    BCHAR   elfStyle[LF_FACESIZE];
} ENUMLOGFONT;
```

Tabelul 1424.2 defineşte membrii structurii *ENUMLOGFONT*.

Membru	Descriere
elfLogFont	Specifică o structură *LOGFONT* care definește atributele unui font.
elfFullName	Specifică un nume unic pentru font. De exemplu, „Compania ABCD de fonturi TrueType Bold Italic Sans Serif".
elfStyle	Specifică stilul unui font. De exemplu „Bold Italic".

Tabelul 1424.2 *Membrii structurii* ***ENUMLOGFONT***.

Ceilalți trei parametrii specifică informații specifice pentru TrueType și tipul fontului (fie *DEVICE_FONTTYPE*, *RASTER_FONTTYPE* sau *TRUETYPE_FONTTYPE* sau combinația lor). Structura *TEXTMETRIC* conține informații de bază despre un font fizic. Toate dimensiunile sunt date în unități logice; asta înseamnă că ele depind de modul curent de mapare al contextului de dispozitiv.

Pentru a înțelege mai bine prelucrările efectuate de funcția *EnumFontFamilies*, analizați programul *Enum_AllFonts.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul utilizează funcția *EnumFontFamilies* pentru completarea unei liste cu numele de fonturi TrueType disponibile. Când utilizatorul selectează opțiunea *Test!* din meniu, programul afișează un șir test în formatul de font selectat curent. Prelucrarea operativă a programului *Enum_AllFonts.cpp* are loc în funcțiile *WndProc*, *EnumFontProc* și *FindFontProc*.

AFIȘAREA DE FONTURI MULTIPLE CU CREATEFUNCTIONINDIRECT

C/C++1425

Programele dumneavoastră pot utiliza funcția *CreateFont* pentru a crea un font. Însă, numărul parametrilor pentru un singur apel la *CreateFont* sunt suficienți pentru a face din *CreateFont* o funcție dificil de utilizat. O alternativă mai bună este funcția *CreateFontIndirect* care creează un font logic care are toate caracteristicile specificate într-o anumită structură. Fontul poate apoi fi selectat ca font curent pentru orice context de dispozitiv. Prototipul funcției *CreateFontIndirect* este următorul:

```
HFONT CreateFontIndirect(
   CONST LOGFONT *lplf // pointer la structura fontului logic
);
```

Parametrul *lplf* indică o structură *LOGFONT* care definește caracteristicile fontului logic. Structura *LOGFONT* definește atributele unui font astfel:

```
typedef struct tagLOGFONT {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
```

```
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Dacă vă uitaţi îndeaproape la structura *LOGFONT* veţi constata că membrii săi corespund direct parametrilor funcţiei *CreateFont* din secţiunea 1423. Membrii acceptă aceleaşi valori ca valorile funcţiei *CreateFont*.

Dacă funcţia *CreateFontIndirect* reuşeşte, valoarea returnată este un identificator pentru un font logic. Dacă eşuează, valoarea returnată de funcţie va fi NULL. Funcţia *CreateFontIndirect* creează un font logic cu toate caracteristicile specificate în structura *LOGFONT*. Când acest font este selectat cu funcţia *SelectObject*, sistemul de mapare al fontului din interfaţa de dispozitiv grafic încearcă să stabilească o corespondenţă între fontul logic şi fontul fizic existent. Dacă nu găseşte o corespondenţă exactă, el va furniza o alternativă ale cărei caracteristici să corespundă cât mai multor caracteristici posibile cerute. Când nu mai aveţi nevoie de font, el trebuie şters cu funcţia *DeleteObject*.

Programul *Enum_AllFonts.cpp* prezentat în secţiunea 1424 utilizează funcţia *CreateFontIndirect*. În loc să invoce *CreateFont* cu paisprezece parametrii, programul va invoca *CreateFontIndirect* cu un singur parametru, ca mai jos:

```
hFont = CreateFontIndirect(&lf);
```

1426 REGĂSIREA CARACTERISTICILOR UNUI DISPOZITIV



Programele dumneavoastră vor utiliza contextul de dispozitiv pentru gestionarea textului şi a graficii în cadrul aplicaţiilor, atât pentru dispozitive de ecran, cât şi pentru dispozitive de imprimare sau plottere. Când programul trebuie să genereze ieşiri la imprimantă sau la plotter, înaintea trimiterii ieşirii la dispozitiv, veţi avea nevoie de informaţii despre caracteristicile dispozitivului. În acest scop, se poate utiliza funcţia *GetDeviceCap*. Această funcţie regăseşte informaţiile specifice ale dispozitivului respectiv. Programul va utiliza funcţia după următorul prototip:

```
int GetDeviceCaps(
    HDC hdc, // identificator pentru contextul de dispozitiv
    int nIndex // indexul caracteristici de investigat
);
```

Parametrul *hdc* identifică contextul de dispozitiv. Parametrul *nIndex* specifică articolul de returnat, care va avea una din valorile descrise în Tabelul 1426.1.

Index	Valoare
DRIVERVERSION	Versiunea driverului de dispozitiv.
TECHNOLOGY	Tehnologia dispozitivului, valorile posibile sunt înscrise în Tabelul 1426.2.
HORZSIZE	Lățimea, în milimetri, a ecranului fizic.
VERTSIZE	Înălțimea, în milimetri, a ecranului fizic.
HORZRES	Lățimea, în pixeli, a ecranului.
VERTRES	Înălțimea, în linii de rastru, a ecranului.
LOGPIXELSX	Numărul de pixeli pe inci logic din lățimea ecranului.
LOGPIXELSY	Numărul de pixeli pe inci logic din înălțimea ecranului.
BITSPIXEL	Numărul de biți consecutivi pentru culoare, pentru fiecare pixel.
PLANES	Numărul de planuri color.
NUMBRUSHES	Numărul de pensule specifice dispozitivului.
NUMPENS	Numărul de creioane specifice dispozitivului.
NUMFONTS	Numărul de fonturi specifice dispozitivului.
NUMCOLORS	Numărul de intrări în tabela de culori a dispozitivului, dacă dispozitivul are o profunzime de culoare mai mare de 8 biți pe pixel. Pentru dispozitivele cu profunzimi de culoare mai mari, se returnează –1.
ASPECTX	Lățimea relativă a unui pixel de dispozitiv utilizat pentru desenarea liniilor.
ASPECTY	Înălțimea relativă a unui pixel de dispozitiv utilizat pentru desenarea liniilor.
ASPECTXY	Lățimea diagonală a pixelului de dispozitiv utilizat pentru desenarea liniilor.
PDEVICESIZE	Rezervat.
CLIPCAPS	Indicator pentru caraceristicile de decupare ale dispozitivului. Dacă dispozitivul poate decupa un dreptunghi, valoarea va fi 1. Altfel, valoarea va fi zero.
SIZEPALETTE	Numărul de intrări în paleta sistemului. Acest index este valid doar dacă driverul dispozitivului activează bitul *RC_PALETTE* din indexul *RASTERCAPS*, fiind disponibil numai dacă driverul este compatibil cu versiunile Windows 3.x sau mai noi.
NUMRESERVED	Numărul de intrări rezervate în paleta sistemului. Acest index este valid doar dacă driverul dispozitivului activează bitul *RC_PALETTE* din indexul *RASTERCAPS*, fiind disponibil numai dacă driverul este compatibil cu versiunile Windows 3.x sau mai noi.
COLORRES	Rezoluția reală a culorii dispozitivului, în biți pe pixeli. Acest index este valid doar dacă driverul dispozitivului activează bitul *RC_PALETTE* din indexul *RASTERCAPS*, fiind disponibil numai dacă driverul este compatibil cu versiunile Windows 3.x sau mai noi.
PHYSICALWIDTH	Pentru dispozitivele de imprimare: lățimea paginii fizice, în unități de dispozitiv. De exemplu, o imprimantă configurată pe 600 dpi la o pagină de 8,5x11 inci are o valoare a lățimii fizice de 5100 de unități de dispozitiv. Remarcați că pagina fizică este aproape întotdeauna mai mare decât zona de imprimare a paginii și niciodată mai mică.

(continuare)

Index	Valoare
PHYSICALHEIGHT	Pentru dispozitivele de imprimare: înălțimea paginii fizice, în unități de dispozitiv. De exemplu, o imprimantă configurată pe 600 dpi la o pagină de 8,5x11 inci are o valoare a înălțimii fizice de 6600 de unități de dispozitiv. Remarcați că pagina fizică este aproape întotdeauna mai mare decât zona de imprimare a paginii și niciodată mai mică.
PHYSICALOFFSETX	Pentru dispozitivele de imprimare: distanța de la marginea stângă a paginii fizice, la marginea stângă a zonei de imprimare, în unități de dispozitiv. De exemplu, o imprimantă configurată pe 600 dpi la o pagină de 8,5x11 inci, care nu poate nu poate imprima în ultimii 0,25 inci din stânga ai paginii, are un deplasament fizic pe orizontală de 150 de unități de dispozitiv.
PHYSICALOFFSETY	Pentru dispozitivele de imprimare: distanța de la marginea de sus a paginii fizice la marginea de sus a zonei de imprimare, în unități de dispozitiv. De exemplu, o imprimantă configurată pe 600 dpi la o pagină de 8,5x11 inci, care nu poate imprima în ultimii 0,5 inci din marginea de sus a paginii, are un deplasament fizic pe verticală de 300 de unități de dispozitiv.
VREFRESH	Pentru dispozitive de afișare: rata de reîmprospătare a dispozitivului, în oscilații pe secundă (Hz), numai sub Windows NT. O valoare a ratei de reîmprospătare pe verticală de 0 sau 1 reprezintă rata de reîmprospătare implicită a dispozitivului hardware de afișare. Rata implicită este de regulă stabilită prin comutatoare pe adaptorul video, pe placa de bază sau prin configurarea programului care nu utilizează funcțiile de afișare Win32, cum ar fi *ChangeDisplaySettings*.
DESKTOPHORZRES	Lățimea, în pixeli, a spațiului de lucru virtual, numai sub Windows NT. Această valoare poate fi mai mare decât *HORZRES* dacă dispozitivul acceptă un spațiu de lucru virtual sau afișări multiple.
DESKTOPVERTRES	Înălțimea, în pixeli, a spațiului de lucru virtual, numai sub Windows NT. Această valoare poate fi mai mare decât *VERTRES* dacă dispozitivul acceptă un spațiu de lucru virtual sau afișări multiple.
BLTALIGNMENT	Alinierea pe orizontală a desenării, preferată de dispozitiv, exprimată ca un multiplu de pixeli, numai sub Windows NT. Pentru performanțe superioare în operațiile de desenare, ferestrele vor trebui aliniate orizontal la un multiplu al acestei valori. Valoarea zero indică faptul că dispozitivul este cu acces accelerat și contextele de dispozitiv pot utiliza orice aliniere.
RASTERCAPS	Valoare care arată caracteristicile rastrului de dispozitiv, prezentate în Tabelul 1426.3.
CURVECAPS	Valoare care arată caracteristicile de linie ale dispozitivului, prezentate în Tabelul 1426.4.
LINECAPS	Valoare care arată caracteristicile de linie ale dispozitivului, prezentate în Tabelul 1426.5.
POLYGONALCAPS	Valoare care arată caracteristicile de poligon ale dispozitivului, prezentate în Tabelul 1426.6.
TEXTCAPS	Valoare care arată caracteristicile de text ale dispozitivului, prezentate în Tabelul 1426.7.

Tabelul 1426.1 *Valorile posibile ale parametrului* ***nIndex***.

Așa cum ați remarcat în Tabelul 1426.1 programul dumneavoastră solicită tehnologia driverului proprie dispozitivului ale cărei valori returnate de funcția *GetDeviceCaps* sunt enumerate în Tabelul 1426.2.

Valoare	Semnificație
DT_PLOTTER	Plotter vectorial.
DT_RASDISPLAY	Dispozitiv de afișare cu rastru.
DT_RASPRINTER	Imprimantă cu rastru.
DT_RASCAMERA	Cameră cu rastru.
DT_CHARSTREAM	Flux de caractere.
DT_METAFILE	Metafișier.
DT_DISPFILE	Fișier pentru afișare.

Tabelul 1426.2 *Indicatorul cu valoarea de returnare a diferitelor tipuri de tehnologii de drivere.*

Este important de observat că în cazul în care parametrul *hdc* identifică un context de dispozitiv al unui metafișier optimizat, tehnologia dispozitivului este similară unui dispozitiv transmis prin referință, ca în cazul funcției *CreateEnhMetaFile*. Pentru a afla dacă este un context de dispozitiv de metafișier optimizat, utilizați funcția *GetObjectType*

Dacă programul dumneavoastră trebuie să identifice caracteristicile de rastru ale dispozitivului respectiv, apelul la *GetDeviceCaps* va include constanta *RASTERCAPS*. Când programele dumneavoastră vor cere caracteristicile de rastru proprii dispozitivului, funcția returnează una sau mai multe din valorile înscrise în Tabelul 1426.3.

Caracteristică	Semnificație
RC_BANDING	Necesită reprezentare prin benzi.
RC_BITBLT	Poate transfera configurații bitmap.
RC_BITMAP64	Poate accepta configurații bitmap mai mari de 64K.
RC_DI_BITMAP	Acceptă funcțiile *SetDIBits* și *GetDIBits*.
RC_DIBTODEV	Acceptă funcția *SetDIBitsToDevice*.
RC_FLOODFILL	Poate efectua operații de umplere continuă (*flood fill*).
RC_GDI20_OUTPUT	Poate accepta caracteristici din Windows 2.0.
RC_PALETTE	Specifică un dispozitiv bazat pe palete.
RC_SCALING	Posibilitate de dimensionare.
RC_STRETCHBLT	Posibilitate de efectuare a funcției *StretchBlt*.
RC_STRETCHDIB	Posibilitate de efectuare a funcției *StretchDIBits*.

Tabelul 1426.3 *Valorile de returnare posibile ale cererii* ***RASTERCAPS***.

În plus față de informațiile despre capacitățile de rasterizare ale dispozitivului, programele vor solicita date despre capacitatea dispozitivului de a desena sau afișa figuri cu linii curbe. Puteți afla posibilitățile unui dispozitiv de a desena linii curbe prin solicitarea *CURVECAPS*, ale cărei valori posibile sunt prezentate în Tabelul 1426.4.

Valoare	Semnificație
CC_NONE	Dispozitivul nu acceptă curbe.
CC_CIRCLES	Dispozitivul poate trasa cercuri.
CC_PIE	Dispozitivul poate desena segmente de cerc.
CC_CHORD	Dispozitivul poate desena arcuri de cerc.
CC_ELLIPSES	Dispozitivul poate desena elipse.
CC_WIDE	Dispozitivul poate desena chenare late.
CC_STYLED	Dispozitivul poate desena chenare stilizate.
CC_WIDESTYLED	Dispozitivul poate desena chenare late și stilizate.
CC_INTERIORS	Dispozitivul poate desena în interior.
CC_ROUNDRECT	Dispozitivul poate desena dreptunghiuri rotunjite.

Tabelul 1426.4 *Valorile returnate la solicitarea* ***CURVECAPS***.

Un dispozitiv poate avea sau nu posibilitatea de a desena linii curbe. Multe dispozitive pot avea posibilități diverse de a trasa linii. Puteți afla caracteristicile unui dispozitiv de a desena linii prin solicitarea *LINECAPS*, ale cărei valori posibile sunt prezentate în Tabelul 1426.5.

Valoare	Semnificație
LC_NONE	Dispozitivul nu acceptă liniile.
LC_POLYLINE	Dispozitivul poate trasa o linie poligonală.
LC_MARKER	Dispozitivul poate desena un marker.
LC_POLYMARKER	Dispozitivul poate desena mai multe elemente marker.
LC_WIDE	Dispozitivul poate desena linii late.
LC_STYLED	Dispozitivul poate linii stilizate.
LC_WIDESTYLED	Dispozitivul poate linii late și stilizate.
LC_INTERIORS	Dispozitivul poate desena în interior.

Tabelul 1426.5 *Valorile returnate la solicitarea* ***LINECAPS***.

În plus față de informațiile despre posibilitățile dispozitivului de a desena linii și cercuri, programele vor solicita date despre capacitatea dispozitivului de a desena sau afișa poligoane, inclusiv dreptunghiuri, trapeze și altele. Puteți afla capacitățile unui dispozitiv de a desena linii curbe, prin solicitarea *PLOLYGONALCAPS*, ale cărei valori posibile sunt prezentate în Tabelul 1426.6.

Valoare	Semnificație
PC_NONE	Dispozitivul nu acceptă poligoanele.
PC_POLYGON	Dispozitivul poate să deseneze poligoane cu hașuri alternative.
PC_RECTANGLE	Dispozitivul poate să deseneze dreptunghiuri.
PC_WINDPOLYGON	Dispozitivul poate să deseneze poligoane cu hașuri ondulate.
PC_SCANLINE	Dispozitivul poate să deseneze o linie de scanare.
PC_WIDE	Dispozitivul poate desena chenare late.

Valoare	Semnificație
PC_STYLED	Dispozitivul poate desena chenare stilizate.
PC_WIDESTYLED	Dispozitivul poate desena chenare late și stilizate.
PC_INTERIORS	Dispozitivul poate desena în interior.

Tabelul 1426.6 *Valorile returnate la solicitarea* ***POLYGONALCAPS***.

Programele dumneavoastră vor cere informații despre posibilitățile dispozitivului de a afișa text. Programele pot afla caracteristicile dispozitivului pentru texte, prin solicitarea *TEXTCAPS*, ale cărei valori posibile sunt prezentate în Tabelul 1426.7.

Bit	Semnificație
TC_OP_CHARACTER	Dispozitivul este capabil de precizie la ieșirile de tip caracter.
TC_OP_STROKE	Dispozitivul este capabil de precizie la ieșirile produse prin acționarea de taste.
TC_CP_STROKE	Dispozitivul este capabil de precizie la decuparea prin acționare de taste.
TC_CR_90	Dispozitivul are posibilitatea de a roti caracterele la 90 de grade.
TC_CR_ANY	Dispozitivul are posibilitatea să rotească oricât caracterele.
TC_SF_X_YINDEP	Dispozitivul poate dimensiona independent pe direcțiile x și y.
TC_SA_DOUBLE	Dispozitivul are posibilitatea de dimensionare a caracterelor duble.
TC_SA_INTEGER	Dispozitivul utilizează multipli de întregi doar pentru dimensionarea caracterelor.
TC_SA_CONTIN	Dispozitivul utilizează orice multipli pentru dimensionarea exactă a caracterelor.
TC_EA_DOUBLE	Dispozitivul poate trasa caractere cu grosime dublă.
TC_IA_ABLE	Dispozitivul poate trasa caractere italice.
TC_UA_ABLE	Dispozitivul poate sublinia caracterele.
TC_SO_ABLE	Dispozitivul poate trasa caractere tăiate.
TC_RA_ABLE	Dispozitivul poate trasa fonturi de rastru.
TC_VA_ABLE	Dispozitivul poate trasa fonturi vectoriale.
TC_RESERVED	Rezervat, trebuie să fie zero.
TC_SCROLLBLT	Dispozitivul nu poate derula prin utilizarea unui transfer de bloc de biți.

Tabelul 1426.7 *Valorile returnate la invocarea* ***TEXTCAPS***.

Pentru a înțelege mai bine prelucrările efectuate de funcția *GetDeviceCaps*, analizați programul *Get_DevC.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Get_DevC.cpp* utilizează funcția *CreateIC* pentru obținerea informațiilor de context ale ecranului. Programul *Get_DevC.cpp* utilizează contextul creat pentru a afla numărul de biți pe pixel și numărul planurilor de culoare ale afișării. Programul *Get_DevC.cpp* utilizează funcția *GetDeviceCapabilities* pentru fiecare determinare. Prelucrarea operativă are loc ca de obicei, în funcția *WndProc*.

1427 UTILIZAREA FUNCȚIEI GETSYSTEMMETRICS PENTRU A ANALIZA O FEREASTRĂ C/C++

Programele dumneavoastră pot lucra cu orice caracteristică a ferestrei, fie ea fereastră părinte, copil sau fereastră control. Una din cele mai obișnuite prelucrări executate de programe este ajustarea dimensiunii de afișare, pentru a corespunde dimensiunii ecranului. Puteți obține aceste informații, împreună cu cele legate de configurările sistemului, prin funcția *GetSystemMetrics*. Funcția *GetSystemMetrics* regăsește configurările pentru diverse sisteme metrice și configurarea de sistem. Sistemele metrice sunt dimensiunile (pe orizontală și verticală) elementelor de afișare din Windows. Toate dimensiunile obținute de *GetSystemMetrics* sunt exprimate în pixeli. Veți implementa funcția *GetSystemMetricis* după următorul prototip:

```
int GetSystemMetrics(int nIndex);
```

Parametrul *nIndex* specifică sistemul metric sau configurația care urmează a fi regăsită. Toate valorile *SM_CX** sunt lățimi. Toate valorile *SM_CY** sunt înălțimi. Interfața Windows API definește aceste valori potrivit listei din Tabelul 1427.1.

Valoare	Semnificație
SM_ARRANGE	Specifică modul în care sistemul a aranjat ferestrele minimizate.
SM_CLEANBOOT	Specifică modul de pornire a sistemului: 0. Boot normal. 1. Boot cu protecție la erori. 2. Boot cu protecție la erori, în rețea.
	Un boot cu protecție la erori sare peste fișierele de startup ale utilizatorului.
SM_CMOUSEBUTTONS	Numărul de butoane ale mouse-ului sau zero, dacă nu este instalat un mouse.
SM_CXBORDER	Lățimea, în pixeli, a chenarului ferestrei. Este echivalent cu *SM_CXEDGE* pentru ferestrele tridimensionale.
SM_CYBORDER	Înălțimea, în pixeli, a chenarului ferestrei. Este echivalent cu *SM_CYEDGE* pentru ferestrele tridimensionale.
SM_CXCURSOR	Lățimea, în pixeli, a unui cursor. Sunt dimensiunile de cursor acceptate de driverul de afișare curent. Sistemul nu poate crea cursoare de alte mărimi.
SM_CYCURSOR	Înălțimea, în pixeli, a unui cursor. Sunt dimensiunile de cursor acceptate de driverul de afișare curent. Sistemul nu poate crea cursoare de alte mărimi.
SM_CXDLGFRAME	La fel ca *SM_CXFIXEDFRAME*.
SM_CYDLGFRAME	La fel ca *SM_CYFIXEDFRAME*.
SM_CXDOUBLECLK	Lățimea, în pixeli, a dreptunghiului din jurul locației primului clic al unei secvențe de dublu clic. Al doilea clic trebuie să intervină în interiorul dreptunghiului pentru ca sistemul să considere cele două clicuri un dublu clic. (Cele două clicuri, de asemenea, trebuie să se succeadă într-un anumit interval de timp.)

Valoare	Semnificație
SM_CYDOUBLECLK	Înălțimea, în pixeli, a dreptunghiului din jurul locației primului clic al unei secvențe de dublu clic. Al doilea clic trebuie să intervină în interiorul dreptunghiului pentru ca sistemul să considere cele două clicuri un dublu clic. (Cele două clicuri, de asemenea, trebuie să se succeadă într-un anumit interval de timp.)
SM_CXDRAG	Lățimea, în pixeli, a dreptunghiului centrat pe punctul de glisare al mouse-ului, pentru a permite deplasări limitate ale identificatorului mouse-ului, înaintea începerii operației de glisare. Aceasta permite utilizatorului să execute clic pe butonul de mouse, apoi să-l elibereze fără să înceapă, neintenționat, operația de glisare.
SM_CYDRAG	Înălțimea, în pixeli, a dreptunghiului centrat pe punctul de glisare al mouse-ului, pentru a permite deplasări limitate ale identificatorului mouse-ului, înaintea începerii operației de glisare. Aceasta permite utilizatorului să execute clic pe butonul de mouse, apoi să-l elibereze fără să înceapă, neintenționat, operația de glisare.
SM_CXEDGE	Lățimea, în pixeli, a unui chenar 3-D. Este valoarea complementară pentru *SM_CXBORDER*.
SM_CYEDGE	Înălțimea, în pixeli, a unui chenar 3-D. Este valoarea complementară pentru *SM_CYBORDER*.
SM_CXFIXEDFRAME	Grosimea exprimată în pixeli, a cadrului din jurul perimetrului unei ferestre care are titlu, dar nu poate fi dimensionat. *SM_CXFIXEDFRAME* este lățimea chenarului orizontal. La fel ca *SM_CXDLGFRAME*.
SM_CYFIXEDFRAME	Grosimea exprimată în pixeli, a cadrului din jurul perimetrului unei ferestre care are titlu, dar nu poate fi dimensionat. *SM_CYFIXEDFRAME* este înălțimea chenarului vertical. La fel ca *SM_CYDLGFRAME*.
SM_CXFRAME	La fel ca *SM_CXSIZEFRAME*.
SM_CYFRAME	La fel ca *SM_CYSIZEFRAME*.
SM_CXFULLSCREEN	Lățimea, în pixeli, a zonei client a unei ferestre de mărimea ecranului. Pentru a obține coordonatele porțiunii ferestrei neascunse de container, apelați funcția *SystemParametersInfo* cu valoarea returnată de *GetSystemMatrics*.
SM_CYFULLSCREEN	Înălțimea, în pixeli, a zonei client a unei ferestre de mărimea ecranului. Pentru a obține coordonatele porțiunii ferestrei neascunse de container, apelați funcția *SystemParametersInfo* cu valoarea *SPI_GETWORKAREA*.
SM_CXHSCROLL	Lățimea, în pixeli, a configurației bitmap săgeată a barei de derulare orizontale.
SM_CYHSCROLL	Înălțimea, în pixeli, a configurației bitmap săgeată a barei de derulare orizontale.

(continuare)

Valoare	Semnificație
SM_CXHTHUMB	Lățimea, în pixeli, a butonului de derulare dintr-o bară de derulare orizontală.
SM_CXICON	Lățimea implicită, în pixeli, a unei pictograme. Aceste valori sunt de regulă 32x32, dar pot varia în funcție de echipamentul hardware instalat. Funcția *LoadIcon* poate să încarce pictograme numai la aceste dimensiuni.
SM_CYICON	Înălțimea implicită, în pixeli, a unei pictograme. Această valoare este de regulă 32x32, dar poate varia în funcție de echipamentul hardware instalat. Funcția *LoadIcon* poate să încarce pictograme numai la aceste dimensiuni.
SM_CXICONSPACING	Dimensiunea X, în pixeli, a unei celule de grilă pentru articolele vizualizate cu opțiunea View | Large Icons. Fiecare element se încadrează într-un dreptunghi cu această dimensiune atunci când este aranjat. Această valoare este întotdeauna mai mare sau egală cu *SM_CXICON*.
SM_CYICONSPACING	Dimensiunea Y, în pixeli, a unei celule de grilă pentru articolele vizualizate cu opțiunea View | Large Icons. Fiecare element se încadrează într-un dreptunghi cu această dimensiune atunci când este aranjat. Această valoare este întotdeauna mai mare sau egală cu *SM_CYICON*.
SM_CXMAXIMIZED	Dimensiunea X implicită, în pixeli, a unei ferestre principale maximizate.
SM_CYMAXIMIZED	Dimensiunea Y implicită, în pixeli, a unei ferestre principale maximizate.
SM_CXMAXTRACK	Lățimea maximă implicită a ferestrei care are titlu și chenare de dimensionare, exprimată în pixeli. Utilizatorul nu poate deplasa cadrul ferestrei la o dimensiune mai mare decât aceasta. O fereastră poate suprascrie această valoare prin prelucrarea mesajului *WM_GETMINMAXINFO*.
SM_CYMAXTRACK	Înălțimea maximă implicită a ferestrei care are titlu și chenare de dimensionare, exprimată în pixeli. Utilizatorul nu poate deplasa cadrul ferestrei la o dimensiune mai mare decât aceasta. O fereastră poate suprascrie această valoare prin prelucrarea mesajului *WM_GETMINMAXINFO*.
SM_CXMENUCHECK	Lățimea, în pixeli, a configurației bitmap implicită pentru semnul de validare din meniu.
SM_CYMENUCHECK	Înălțimea, în pixeli, a configurației bitmap implicită pentru semnul de validare din meniu.
SM_CXMENUSIZE	Lățimea, în pixeli, a butoanelor din bara de meniu, cum ar fi butonul de închidere a unei ferestre copil MDI.
SM_CYMENUSIZE	Înălțimea, în pixeli, a butoanelor din bara de meniu, cum ar fi butonul de închidere a unei ferestre copil MDI.
SM_CXMIN	Lățimea minimă, în pixeli, a unei ferestre.
SM_CYMIN	Înălțimea minimă, în pixeli, a unei ferestre.

Valoare	Semnificație
SM_CXMINIMIZED	Lățimea, în pixeli, a unei ferestre normale minimizate.
SM_CYMINIMIZED	Înălțimea, în pixeli, a unei ferestre normale minimizate.
SM_CXMINSPACING	Dimensiunea X, în pixeli, a celulei de grilă pentru ferestrele minimizate. Fiecare fereastră minimizată se încadrează într-un dreptunghi de dimensiunile acestea când este aranjată. Această valoare este întotdeauna mai mare sau egală cu *SM_CXMINIMIZED*.
SM_CYMINSPACING	Dimensiunea Y, în pixeli, a celulei de grilă pentru ferestrele minimizate. Fiecare fereastră minimizată se încadrează într-un dreptunghi de dimensiunile acestea când este aranjată. Această valoare este întotdeauna mai mare sau egală cu *SM_CYMINIMIZED*.
SM_CXMINTRACK,	Lățimea minimă de urmărirea în operația de glisare cu mouse-ul a unei ferestre. Utilizatorul nu poate glisa cadrul ferestrei la o mărime mai mică decât aceasta. O fereastră poate suprascrie această valoare prin prelucrarea mesajului *WM_GETMINMAXINFO*.
SM_CYMINTRACK	Înălțimea minimă de urmărire în operația de glisare cu mouse-ul a unei ferestre. Utilizatorul nu poate glisa cadrul ferestrei la o mărime mai mică decât aceasta. O fereastră poate suprascrie această valoare prin prelucrarea mesajului *WM_GETMINMAXINFO*.
SM_CXSCREEN	Lățimea ecranului, în pixeli.
SM_CYSCREEN	Înălțimea ecranului, în pixeli.
SM_CXSIZE	Lățimea, în pixeli, a unui buton într-o bară de titlu.
SM_CYSIZE	Înălțimea, în pixeli, a unui buton într-o bară de titlu.
SM_CXSIZEFRAME	Grosimea exprimată în pixeli, a cadrului din jurul perimetrului unei ferestre pe care utilizatorul o poate redimensiona. *SM_CXSIZEFRAME* este lățimea chenarului orizontal. La fel ca *SM_CXFRAME*.
SM_CYSIZEFRAME	Grosimea exprimată în pixeli, a cadrului din jurul perimetrului unei ferestre pe care utilizatorul o poate redimensiona. *SM_CYSIZEFRAME* este înălțimea chenarului vertical. La fel ca *SM_CYFRAME*.
SM_CXSMICON	Dimensiunea X, în pixeli, recomandată pentru o pictogramă mică. Pictogramele mici apar de regulă în titlurile ferestrelor și în vizualizările cu opțiunea View | Small Icons.
SM_CYSMICON	Dimensiunea Y, în pixeli, recomandată pentru o pictogramă mică. Pictogramele mici apar de regulă în titlurile ferestrelor și în vizualizările cu opțiunea View | Small Icons.
SM_CXSMSIZE	Dimensiunea X, în pixeli, a butoanelor mici din titlu.
SM_CYSMSIZE	Dimensiunea Y, în pixeli, a butoanelor mici din titlu.
SM_CXVSCROLL	Lățimea, în pixeli, a barei de derulare verticală.

(continuare)

Valoare	Semnificație
SM_CYVSCROLL	Înălțimea, în pixeli, a configurației bitmap săgeată dintr-o bară de derulare verticală.
SM_CYCAPTION	Înălțimea, în pixeli, a zonei normale de titlu.
SM_CYKANJIWINDOW	Pentru versiunile de Windows cu seturi de caractere pe dublu-octet, înălțimea, în pixeli, a ferestrei *Kanji*, în partea de jos a ecranului.
SM_CYMENU	Înălțimea, în pixeli, a unei bare de meniu pe un singur rând.
SM_CYSMCAPTION	Înălțimea, în pixeli, a unui titlu mic.
SM_CYVTHUMB	Înălțimea, în pixeli, a unui buton de derulare dintr-o bară de derulare verticală.
SM_DBCSENABLED	TRUE sau diferită de zero, dacă o versiune de *USER.EXE* cu set dublu de caractere (DBCS) este instalată; în caz contrar, FALSE sau zero.
SM_DEBUG	TRUE sau diferită de zero, dacă o versiune de *USER.EXE* pentru depanare este instalată; în caz contrar, FALSE sau zero.
SM_MENUDROPALIGNMENT	TRUE sau diferită de zero, dacă meniurile derulante sunt aliniate la dreapta în raport cu articolul din bara de meniu corespunzător. FALSE sau zero dacă este aliniat la stânga.
SM_MIDEASTENABLED	TRUE, dacă sistemul este activat pentru limbile ebraică/arabă.
SM_MOUSEPRESENT	TRUE sau diferit de zero, dacă mouse-ul este instalat; FALSE sau diferit de zero în caz contrar.
SM_MOUSEWHEELPRESENT	Numai sub Windows NT: TRUE sau diferit de zero, dacă este instalat un mouse cu bilă; FALSE sau diferit de zero în caz contrar.
SM_NETWORK	Funcția returnează o valoare cu bitul cel mai puțin semnificativ activat, dacă este prezentă o rețea. În caz contrar, bitul este dezactivat. Ceilalți biți sunt rezervați pentru utilizări viitoare.
SM_PENWINDOWS	TRUE sau diferit de zero, dacă este instalat MS Windows pentru extensiile Pen; zero sau FALSE, în caz contrar.
SM_SECURE	TRUE dacă este prezent sistemul de securitate. Altfel, FALSE.
SM_SHOWSOUNDS	TRUE sau diferit de zero, dacă utilizatorul solicită aplicației să prezinte vizual informațiile în situații când, altfel, le-ar fi prezentat în formă sonoră. Altfel, FALSE sau zero.
SM_SLOWMACHINE	TRUE când calculatorul are un procesor lent și FALSE, în caz contrar.
SM_SWAPBUTTON	TRUE sau diferit de zero, dacă sunt inversate operațiile de dreapta-stânga ale mouse-ului. FALSE sau zero în caz contrar.

Tabelul 1427.1 *Sistemele metrice posibile și configurații de sistem.*

Dacă funcția *GetSystemMetrics* reușește, valoarea returnată este sistemul metric sau configurația de sistem. Dacă funcția eșuează, valoarea returnată este zero,

Sistemele metrice pot varia de la un sistem de afișare la altul. Valoarea *SM_ARRANGE* specifică modul în care sistemul aranjează ferestrele minimizate și constă într-o poziție de început și o direcție. Pozițiile de început sunt prezentate în Tabelul 1427.2.

Valoare	Semnificație
ARW_BOTTOMLEFT	Începe în colțul din stânga jos al ecranului (poziția implicită).
ARW_BOTTOMRIGHT	Începe în colțul din dreapta jos al ecranului. Echivalent cu *ARW_STARTRIGHT*.
ARW_HIDE	Ascunde ferestrele minimizate prin mutarea lor în afara zonei vizibile a ecranului.
ARW_TOPLEFT	Începe în colțul din stânga sus al ecranului. Echivalent cu *ARV_STARTTOP*.
ARW_TOPRIGHT	Începe în colțul din dreapta sus al ecranului. Echivalent cu *ARW_STARTTOP | SRW_STARTRIGHT*.

Tabelul 1427.2 *Valorile de început ale poziției.*

Direcția în care sistemul aranjează ferestrele minimizate poate lua una din valorile prezentate în Tabelul 1427.3.

Valoare	Semnificație
ARW_DOWN	Aranjare pe verticală, de sus în jos.
ARW_LEFT	Aranjare pe orizontală, de la stânga la dreapta.
ARW_RIGHT	Aranjare pe orizontală, de la dreapta la stânga.
ARW_UP	Aranjare pe verticală, de jos în sus.

Tabelul 1427.3 *Valorile de aranjare posibile.*

Secțiunea 1428 oferă detalii asupra utilizării sistemelor metrice în timpul prelucrării.

UTILIZAREA FUNCȚIEI GETSYSTEMMETRICS

C/C++1428

Așa cum ați învățat în secțiunea 1427, programele dumneavoastră pot utiliza *GetSystemMetrics* pentru a obține informații despre sistemele metrice curente ale sistemului Windows pe orice calculator. Pe măsură ce programele dumneavoastră devin mai complexe și dezvoltați aplicații care vor rula pe diferite calculatoare, devine importantă aflarea sistemului metric înaintea generării elementelor de afișare, deoarece vă asigură că afișarea va arăta corespunzător, atât pe ecranele cu 640x480, cât și pe cele 1024x768. Programul *Get_Sysm.cpp*, de exemplu, conținut pe CD-ROM-ul care însoțește cartea de față, utilizează funcția *GetSystemMetrics* pentru a enumera starea curentă a dimensiunilor ferestrei, ecranului și late informații. Când utilizatorul selectează opțiunea *Test!*, programul afișează informațiile în fereastră.

OBȚINEREA CONTEXTULUI DE DISPOZITIV AL UNEI FERESTRE ÎNTREGI

C/C++1429

Programele dumneavoastră operează cu contextul de dispozitiv în zona client a ferestrei. Se vor ivi însă ocazii când programul va necesita un context de dispozitiv pentru întreaga

fereastră. În această situaţie, poate fi utilizată funcţia *GetWindowDC* care obţine un context de dispozitiv (DC) pentru întreaga fereastră, inclusiv bara de titluri, meniurile şi barele de derulare. Contextul de dispozitiv de fereastră permite operaţii de trasare în toate zonele ferestrei, pentru că originea contextului de dispozitiv este colţul de stânga sus al ferestrei, nu al zonei client.

Funcţia *GetWindowDC* alocă atributele implicite contextului de dispozitiv al ferestrei, de fiecare dată când regăseşte contextul de dispozitiv. Deoarece *GetWindowDC* alocă atributele implicite, atributele anterioare se pierd. Veţi utiliza funcţia conform următorului prototip:

```
HDC GetWindowDC(
    HWND hWnd // identificator pentru fereastra
);
```

Parametrul *hWnd* identifică fereastra al cărei context de dispozitiv trebuie regăsit. Dacă funcţia *GetWindowDC* reuşeşte, valoarea returnată este identificatorul contextului de dispozitiv al ferestrei specificate, iar dacă eşuează, valoarea returnată este NULL, indicând o eroare sau un parametru *hWnd* nevalid.

Funcţia *GetWindowDC* este utilizată pentru efecte speciale de trasare în zona ne-client a ferestrei. Trasarea în zonele ne-client ale oricărei ferestre nu este recomandabilă. Puteţi utiliza funcţia *GetSystemMetrics* pentru regăsirea dimensiunilor diferitelor părţi ale zonei ne-client, cum ar fi bara de titlu, bara de meniu şi barele de derulare. După încheierea trasării, trebuie apelată funcţia *ReleaseDC* pentru eliberarea contextului de dispozitiv. Neînlăturarea contextului de dispozitiv al ferestrei poate avea consecinţe grave asupra operaţiilor de trasare ulterioare ale aplicaţiei.

Pentru a înţelege mai bine prelucrările efectuate de funcţia *GetWindowDC*, analizaţi programul *GetWinDC.cpp* conţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *GetWinDC* foloseşte funcţia *GetWindowDC* pentru regăsirea contextului de dispozitiv al întregii ferestre. Apoi utilizează funcţia *GetSystemMetrics* pentru determinarea dimensiunilor cadrului şi barei de titlu. În final, utilizează contextul de dispozitiv al întregii ferestre pentru a trasa un model în bara de titlu a ferestrei. Prelucrarea operativă are loc în funcţia *WndProc*.

1430 ELIBERAREA CONTEXTELOR DE DISPOZITIV C/C++

Eliberarea unui obiect după terminarea prelucrării lui de către program face parte integrantă din programarea în C++. Lucrul cu contextele de dispozitiv nu face excepţie. Funcţia *ReleaseDC* eliberează un context de dispozitiv pentru utilizarea lui în alte aplicaţii. Efectul funcţiei *ReleaseDC* depinde de tipul contextului de dispozitiv. Funcţia eliberează numai contexte de dispozitiv obişnuite şi de fereastră şi nu are efect asupra contextelor de dispozitiv de clasă sau private. Veţi implementa funcţia *ReleaseDC* potrivit prototipului de mai jos:

```
int ReleaseDC(
    HWND hWnd, // identificator pentru fereastra
    HDC hDC // identificator pentru contextul de dispozitiv
);
```

Parametrul *hWnd* identifică fereastra al cărei context de dispozitiv trebuie eliberat. Parametrul *hDC* identifică contextul de dispozitiv care urmează a fi eliberat. Valoarea returnată de funcţia *ReleaseDC* precizează dacă respectivul context de dispozitiv a fost sau nu eliberat.

În cazul în care contextul de dispozitiv este eliberat, funcția returnează 1. Dacă, dimpotrivă, contextul nu este eliberat, funcția returnează 0.

Aplicațiile trebuie să apeleze funcția *ReleaseDC* după fiecare apel la funcția *GetWindowDC* și după fiecare apel la funcția *GetDC*, care regăsește un context de dispozitiv obișnuit. Aplicațiile nu pot utiliza funcția *ReleaseDC* pentru eliberarea contextului de dispozitiv produs prin apelarea funcției *CreateDC*; în schimb, trebuie să utilizeze funcția *DeleteDC*.

OBȚINEREA UNUI IDENTIFICATOR DE FEREASTRĂ DIN CONTEXTUL DE DISPOZITIV

C/C++1431

Programele dumneavoastră vor efectua prelucrări generalizate în orice context de dispozitiv dat. Însă, puteți să evitați aceste prelucrări în contextul de dispozitiv asociat unei ferestre date. Cu funcția *WindowFromDC*, programul dumneavoastră poate converti un context de dispozitiv la un identificator de fereastră. Funcția returnează identificatorul ferestrei asociate cu contextul de dispozitiv de afișare dat (DC). Funcțiile de ieșire care utilizează acest context de dispozitiv trasează în această fereastră a cărui identificator este returnat de *WindowFromDC*. Veți implementa funcția *WindowFromDC* potrivit prototipului prezentat mai jos:

```
HWND WindowFromDC(
    HDC hDC // identificator de fereastra
);
```

Parametrul *hDC* identifică contextul de dispozitiv din care va fi regăsit identificatorul pentru fereastra asociată. Dacă funcția reușește, valoarea returnată este identificatorul ferestrei asociate cu respectivul context de dispozitiv de afișare. Dacă funcția eșuează, valoarea returnată va fi NULL.

CE ESTE UN BITMAP DEPENDENT DE DISPOZITIV

C/C++1432

Un bitmap este un bloc de date pe care programele îl pot trimite direct la un dispozitiv, cum ar fi un display video. Puteți să vă gândiți la bitmap ca la un mijloc de stocare a datelor cu pixeli de pe ecran direct într-un buffer de memorie. Trasarea unui bitmap pe ecran este mult mai rapidă decât utilizarea funcțiilor de interfață cu dispozitivele grafice, cum ar fi *Rectangle* sau *LineTo*. Neajunsurile unui bitmap constau în consumul mare de memorie și de spațiu pe disc și nu se dimensionează bine mai ales dacă includ și text. Dimensionarea unui bitmap duce la deprecierea calității și distorsiunea textului.

Windows pune la dispoziție două tipuri de bitmap: bitmap dependent de dispozitiv și bitmap independent de dispozitiv. Un bitmap dependent de dispozitiv este un format mai vechi, care așa cum sugerează și numele, este mai puțin flexibil decât cel independent de dispozitiv. Toate aplicațiile Win32 pe care le scrieți e bine să folosească blocuri bitmap independente de dispozitiv, deoarece Windows furnizează majoritatea funcțiilor care prelucrează blocuri bitmap dependente de dispozitiv numai din motive de compatibilitate cu aplicațiile scrise pe 16 biți.

Pentru producerea blocurilor bitmap se utilizează programe de desenare ca Microsoft Paint sau un editor de bitmap component al unui mediu de dezvoltare integrat (Borland C++ 5.2

sau MS Visual C++ 5.0). Blocul bitmap este stocat pe disc cu extensia .BMP. Windows îl va salva ca bitmap independent de dispozitiv și îl va converti într-un bitmap dependent de dispozitiv în momentul în care programul va invoca *LoadBitmap*. Puteți precede numele de fișier de bitmap cu cuvântul cheie *BITMAP* pentru adăuga un bitmap într-un fișier de resurse, cum se prezintă mai jos:

```
pen BITMAP pen.bmp
```

Pentru a înțelege mai bine prelucrările efectuate de programele dumneavoastră pentru lucrul cu blocuri bitmap, analizați programul *Pen_BMP.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Pen_BMP* încarcă un bitmap creion (pen), îl plasează într-un context de dispozitiv de memorie și apoi desenează cu el în contextul de dispozitiv de afișare. Următorul fragment de cod din fișierul *Pen_BMP.cpp* arată partea operativă a prelucrării:

```
HBITMAP hBitmap;
HDC     hDC;
HDC     hMemDC;
// Incarca un bitmap in memorie
hBitmap = LoadBitmap( hInst, "pen" );
// Deseneaza un bitmap in MemDC si apoi pe ecran
hDC = GetDc( hWnd );
hMemDC = CreateCompatibleDC( hDC);
SelectObject( hMemDC, hBitmap );
BitBlt( hDC, 10, 10, 60, 60, hMemDC, 0, 0, SRCCOPY );
DeleteDC( hMemDC );
ReleaseDC( hWnd, hDC );
DeleteObject( hBitmap );
```

Formatul bitmap dependent de dispozitiv operează eficient în copierea secțiunilor ecranului în memorie și alipirea acestor secțiuni înapoi în alte localizări ale ecranului. Când însă aplicația trebuie să salveze datele într-un fișier pe disc și apoi să le afișeze pe un alt dispozitiv, formatul bitmap dependent de dispozitiv eșuează. Formatul bitmap dependent de dispozitiv presupune că va fi afișat întotdeauna pe un dispozitiv asemănător cu cel pe care a fost inițial produs și culorile sunt similare pe cel de-al doilea dispozitiv cu cele de pe primul. Neajunsul este că atunci când afișați blocul bitmap pe un alt dispozitiv, culorile pot fi diferite. Blocul bitmap independent de dispozitiv elimină problemele formatului bitmap dependent de dispozitiv.

1433 Bitmap independent de dispozitiv



Așa cum ați învățat în secțiunea 1432, formatul bitmap dependent de dispozitiv are anumite limite. Formatul independent de dispozitiv nu mai are aceste limitări. Cea mai importantă diferență între un bitmap dependent de dispozitiv și un bitmap independent de dispozitiv este că acesta din urmă conține o tabelă cuprinzând culorile ce vor fi folosite de bitmap. Antetul de bitmap este, de asemenea, mai complex în cazul formatului independent de dispozitiv. Spre deosebire de cel dependent de dispozitiv, un bitmap independent de dispozitiv nu este un obiect grafic, ci mai curând un format de date. Aplicațiile nu pot selecta un bitmap independent de dispozitiv într-un context de dispozitiv. Formatul unui bitmap

independent de dispozitiv constă în trei secțiuni de date separate. Prima secțiune a fișierului de format bitmap independent de dispozitiv este structura *BITMAPINFOHEADER*, definit în Windows API în modul arătat mai jos:

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

Structura *BITMAPINFOHEADER* conține membrii prezentați în Tabela 1433.1.

Nume membru	Descriere
biSize	Specifică numărul de octeți cerut de structură.
biWidth	Specifică lățimea blocului bitmap, în pixeli.
biHeight	Specifică înălțimea blocului bitmap, în pixeli. Dacă *biHeight* este pozitiv, blocul bitmap este de tip jos-sus, iar originea lui este colțul din stânga jos. Dacă *biHeight* este negativ, blocul bitmap este de tip sus-jos, iar originea sa este colțul din stânga sus.
biPlanes	Specifică numărul de planuri ale dispozitivului țintă. Această valoare trebuie fixată la 1.
biBitCount	Specifică numărul de biți pe pixel. Această valoare trebuie să fie 1, 4, 8, 16, 24 sau 32.
biCompression	Specifică tipul de comprimare al unui bitmap jos-sus comprimat (blocurile bitmap independente de dispozitiv sus-jos nu pot fi comprimate). Poate lua una din valorile înscrise în Tabelul 1433.2.
biSizeImage	Specifică dimensiunea imaginii, în biți. Poate fi fixată la zero pentru blocurile bitmap *BI_RGB*.
biXPelsPerMeter	Specifică rezoluția orizontală, în pixeli pe metru, a dispozitivului țintă al blocului bitmap. Aplicațiile pot utiliza această valoare pentru a selecta un bitmap dintr-un grup de resurse care corespund cel mai bine caracteristicilor dispozitivului curent.
biYPelsPerMeter	Specifică rezoluția pe verticală, în pixeli pe metru, a dispozitivului țintă al blocului bitmap.
biClrUsed	Specifică numărul de indici de culoare într-o tabelă de culoare, utilizate la momentul curent de către bitmap. Dacă această valoare este zero, blocul bitmap utilizează numărul maxim de culori corespunzător valorii membrului *biBitCount* pentru modul de comprimare specificat prin *biCompression*.

(continuare)

Nume membru	Descriere
	Dacă *biClrUsed* nu este zero și membrul *biBitCount* este mai mic decât 16, *biClrUsed* specifică numărul real de culori la care au acces motorul grafic sau driverul dispozitivului. Dacă *biBitCount* este egal sau mai mare decât 16, atunci membrul *biClrUsed* specifică dimensiunea tabelei de culoare utilizate pentru optimizarea performanței paletei de culori din Windows. Dacă *biBitCount* este egal cu 16 sau 32, paleta de culoare optimă începe imediat urmând celor trei măști dublu-cuvânt. Dacă blocul bitmap este un bitmap împachetat (în care matricea bitmap urmează imediat antetului *BITMAPINFO* și care este referențiat de un singur pointer), membrul *biClrUsed* trebuie să ia fie valoarea 0, fie valoarea reală a tabelei de culoare.
biClrImportant	Specifică numărul de indici de culoare considerați importanți pentru afișarea blocului bitmap. Dacă valoarea este zero, toate culorile sunt importante,

Tabelul 1433.1 *Membrii structurii **BITMAPINFOHEADER**.*

Așa cum ați aflat din Tabelul 1433.1, puteți produce blocuri bitmap jos-sus comprimate. Când programul încarcă un bitmap comprimat, trebuie să afle valoarea membrului *bitCompressed* pentru obținerea tipului de comprimare. Tabela 1433.2 prezintă valorile posibile ale membrului *biCompressed*.

Valoare	Descriere
BI_RGB	Format necomprimat.
BI_RLE8	Un format RLEB (run-length encoded bitmap – un format de comprimare a imaginii care codifică secvențe de pixeli identici) pentru blocuri bitmap cu 8 biți pe pixel. Formatul de comprimare este un format dublu-octet care constă într-un octet de contor urmat de un octet cu indicele culorii.
BI_RLE4	Un format RLEB pentru blocurile bitmap cu 4 biți pe pixel. Formatul de comprimare este un format dublu-octet care constă într-un octet de contor urmat de doi indici dublu-cuvânt de culoare.
BI_BITFIELDS	Precizează că blocul bitmap nu este comprimat și că tabela de culoare conține trei măști de culoare dublu-cuvânt care conțin componentele de roșu, verde și albastru ale fiecărui pixel. Valid numai cu utilizarea blocurilor bitmap de 16 sau 32 de biți pe pixel.

Tabela 1433.2 *Valorile posibile ale membrului **biCompressed**.*

Structura *BITMAPINFO* combină structura *BITMAPINFOHEADER* cu tabela de culoare pentru a furniza definiția completă a dimensiunilor și culorilor unui bitmap independent de dispozitiv. O aplicație ar trebui să folosească informațiile stocate în membrul *biSize* pentru identificarea tabelei de culoare dintr-o structură *BITMAPINFO*, în modul de mai jos:

```
pColor = ((LPSTR)pBitmapInfo +
   (WORD)(pBitmapInfobmiHeader.biSize));
```

Windows acceptă formate de comprimare a blocuilor bitmap care își definesc culorile cu opt sau patru biți pe pixel. Comprimarea reduce spațiul de pe disc sau din memorie necesar pentru bitmap. În Tabelul 1433.2 ați aflat de cele trei formate de comprimare.

Când membrul *biCompression* este *BI_RLE8*, blocul bitmap este comprimat folosind un format RLEB pentru un bitmap cu 8 biți pe pixel. Acest format poate fi comprimat în modul codificat sau absolut. Ambele moduri pot apărea în orice loc în același bitmap.

Modul codificat constă în doi octeți: primul octet specifică numărul de pixeli consecutivi ce vor fi desenați folosind indicele de culoare conținut în al doilea octet. În plus, programul care a produs blocul bitmap poate fixa primul octet al perechii pe zero, pentru indicarea unui caracter de escape care denotă un sfârșit de linie, un sfârșit de bitmap sau delta (adică o modificare). Interpretarea pentru escape depinde de valoarea octetului al doilea al perechii, care poate lua una din valorile descrise în Tabelul 1433.3.

Valoare	Semnificație
0	Sfârșit de linie.
1	Sfârșit de bitmap.
2	Delta. Cei doi octeți care urmează după escape conțin valori fără semn care indică deplasamentele orizontale și verticale ale următorului pixel din poziția curentă.

***Tabelul 1433.3** Valorile posibile ale celui de al doilea octet din pereche.*

Pe de altă parte, în modul absolut, primul octet este zero și al doilea ia o valoare în intervalul 03H la FFH. Al doilea octet reprezintă numărul de octeți care urmează, fiecare conținând indicele de culoare pentru un singur pixel. Când al doilea octet este 2 sau mai puțin, escape are aceeași semnificație cu modul codificat. În modul absolut, programul care creează trebuie să alinieze fiecare rând din blocul bitmap independent de dispozitiv, în limita unui cuvânt.

După structura *BITMAPINFOHEADER*, un bitmap independent de dispozitiv conține tabela de culoare. Tabela de culoare este un set de date ale structurii *RGBQUAD* care conține culoarea *RGB* pentru fiecare culoare utilizată de bitmap. Structura *RGBQUAD* descrie o culoare cu atributele intensității relative de roșu, verde și albastru. Numărul intrărilor în structura *RGBQUAD* va corespunde numărului de opțiuni de culoare din bitmap. Interfața Windows API construiește structura *RGBQUAD* în modul descris mai jos:

```
typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
} RGBQUAD;
```

Structura *RGBQUAD* conține membrii descriși în Tabelul 1433.4.

Membru	Descriere
rgbtBlue	Specifică intensitatea de albastru a culorii
rgbtGreen	Specifică intensitatea de verde a culorii
rgbtRed	Specifică intensitatea de roșu a culorii
rgbReserved	Windows rezervă acest membru; trebuie să fie zero

***Tabelul 1433.4** Membrii structurii **RGBQUAD**.*

Restul fișierului bitmap independent de dispozitiv conține datele reale cu pixelii pentru bitmap. Veți învăța mai mult despre producerea și afișarea formatelor bitmap dependente și independente de dispozitiv, în cursul unor secțiuni următoare.

1434 CREAREA BLOCURILOR BITMAP



Unul dintre cele trei tipuri native de fișiere grafice specifice pentru Windows API este fișierul bitmap. În programele dumneavoastră puteți produce blocuri bitmap ce pot fi afișate în ferestre. În acest scop, veți utiliza funcția *CreateBitmap* care produce un bitmap cu lățimea, înălțimea și formatul de culoare (planuri de culoare și biți pe pixel) specificate. Veți implementa funcția *CreateBitmap* potrivit prototipului prezentat mai jos:

```
HBITMAP CreateBitmap(
    int nWidth,           // latime bitmap, in pixeli
    int nHeight,          // inaltime bitmap, in pixeli
    UINT cPlanes,         // numar planuri de culoare utilizate
    UINT cBitsPerPel,     // numar de biti pentru indentificarea
                          // culorii
    CONST VOID *lpvBits // pointer la o matrice cu date de
                          // culoare
);
```

Când programul apelează funcția *CreateBitmap*, aceasta va aștepta parametrii prezentați în Tabelul 1434.1.

Parametri	Descriere
nWidth	Specifică lățimea blocului bitmap, în pixeli.
nHeight	Specifică înălțimea blocului bitmap, în pixeli.
cPlanes	Specifică numărul de planuri de culoare utilizate de dispozitiv.
cBitsPerPel	Specifică numărul de biți necesari identificării culorii unui singur pixel.
lpvBits	Indică o matrice de date pentru stabilirea culorilor într-un dreptunghi de pixeli. Fiecare linie de scanare din dreptunghi va trebui să respecte aliniamentul de tip cuvânt (liniile scanate care nu se aliniază în limita unui cuvânt sunt completate cu zero). Dacă acest parametru este NULL, noul bitmap este nedefinit.

***Tabelul 1434.1** Parametrii funcției **CreateBitmap**.*

Dacă funcția *CreateBitmap* reușește, valoarea returnată este identificatorul de bitmap. Dacă funcția eșuează, valoarea returnată va fi NULL.

După producerea unui bitmap, el poate fi selectat într-un context de dispozitiv, prin apelarea funcției *SelectObject*. Deoarece funcția poate fi folosită pentru producerea unro blocuri bitmapr color, pentru creșterea performanțelor se recomandă *CreateBitmap* pentru producerea blocurilor bitmap monocrome și funcția *CreateCompatibleBitmap*, pentru cele color. Când un bitmap returnat de *CreateBitmap* este selectat într-un context de dispozitiv, Windows trebuie să asigure ca blocul bitmap să corespundă cu formatul contextului de dispozitiv în care este selectat. Deoarece *CreateCompatibleBitmap* preia un context de dispozitiv, această funcție returnează un bitmap care are același format ca și contextul de dispozitiv specificat. De aceea, următoarele apeluri la *SelectObject* sunt mai rapide decât în cazul blocurilor bitmap returnate de *CreateBitmap*.

Dacă blocul bitmap este monocrom, zerourile reprezintă culoarea de prim plan, iar cifrele unu, culoarea de fundal a contextului de dispozitiv destinație. Dacă o aplicație fixează

parametrii *nWidth* și *nHeight* la zero, *CreateBitmap* returnează identificatorul unui bitmap monocrom de tip 1-1. Când nu mai este nevoie de bitmap, apelați funcția *DeleteObject* pentru a-l șterge.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateBitmap*, analizați programul *Create_Bitmap.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Create_Bitmap.cpp* produce un bitmap monocrom la selecția opțiunii *Test!* din meniu. Programul selectează blocul bitmap într-un context de dispozitiv de memorie și desenează un dreptunghi și o elipsă în bitmap. Programul afișează apoi noul bitmap. Prelucrarea operativă are loc în funcția *WndProc*.

AFIȘAREA BLOCURILOR BITMAP

C/C++1435

Așa cum ați învățat în secțiunea 1434, programele dumneavoastră trebuie să „trimită" un bitmap la contextul de dispozitiv de afișare pentru a-l afișa. În programul *Create_Bitmap.cpp* ați produs mai întâi blocul bitmap, apoi l-ați adăugat contextului de dispozitiv pentru afișare. De regulă, programele dumneavoastră vor utiliza funcția *BitBlt* pentru afișarea unui bitmap. Funcția *BitBlt* efectuează un transfer de blocuri de biți cu datele de culoare care corespunde unui dreptunghi de pixeli din contextul de dispozitiv sursă specificat, într-un context de dispozitiv destinație. Mai întâi trebuie selectat blocul bitmap într-un context de dispozitiv de memorie (creat cu *CreateCompatibleDC*). Apoi, veți apela funcția *BitBlt* pentru a copia blocul bitmap în contextul de dispozitiv read. Deoarece procedura reală *BitBlt* de copiere folosește operații de rastru, trebuie să aflați valoarea *RASTERCAPS* a unui dispozitiv înainte de a executa funcția *BitBlt* pe acel dispozitiv. Veți implementa funcția *BitBlt* potrivit prototipului prezentat mai jos:

```
BOOL BitBlt(
    HDC hdcDest,        // identificator pentru contextul de
                        // dispozitiv destinatie
    int nXDest,         // coordonata x a coltului stanga sus al
                        // dreptunghiului de destinatie
    int nYDest,         // coordonata y a coltului stanga sus al
                        // dreptunghiului de destinatie
    int nWidth,         // latimea dreptunghiului de destinatie
    int nHeight,        // inaltimea dreptunghiului de destinatie
    HDC hdcSrc,         // identificator pentru contextul de
                        // dispozitiv sursa
    int nXSrc,          // coordonata x a coltului stanga sus al
                        // dreptunghiului sursa
    int nYSrc,          // coordonate y a coltului stanga sus al
                        // dreptunghiului sursa
    DWORD dwRop         // cod de operare rastru
);
```

Funcția *BitBlt* acceptă parametrii prezentați în Tabelul 1435.1.

Parametru	Descriere
hdcDest	Identifică contextul de dispozitiv destinaţie.
nXDest	Conţine coordonata x logică a colţului din stânga sus al dreptunghiului de destinaţie.
nYDest	Conţine coordonata y logică a colţului din stânga sus al dreptunghiului de destinaţie.
nWidth	Specifică lăţimea logică a dreptunghiurilor sursă şi destinaţie.
nHeight	Specifică înălţimea logică a dreptunghiurilor sursă şi destinaţie.
hdcSrc	Identifică sursa contextului de dispozitiv.
nXSrc	Conţine coordonata x logică a colţului din stânga sus al dreptunghiului sursă.
nYSrc	Conţine coordonata y logică a colţului din stânga sus al dreptunghiului sursă.
dwRop	Specifică un cod de operare cu rastru. Aceste coduri definesc modul în care datele de culoare pentru dreptunghiul sursă vor fi combinate cu datele de culoare pentru dreptunghiul destinaţie, în vederea producerii culorii finale.

Tabelul 1435.1 *Parametrii funcţiei* ***BitBlt***.

Programele vor efectua anumite operaţii cu rastru pentru blocurile bitmap plasate în contextele de dispozitiv. Tabelul 1435.2 descrie câteva coduri de operaţii cu rastru obişnuite.

Valoare	Descriere
BLACKNESS	Umple dreptunghiul destinaţie folosind culoarea asociată cu indicele 0 din paleta fizică. (Această culoare este negru în paleta fizică implicită.)
DSTINVERT	Inversează dreptunghiul de destinaţie.
MERGECOPY	Uneşte culorile dreptunghiului sursă cu modelul specificat, folosind operatorul boolean AHD (ŞI).
MERGEPAINT	Uneşte culorile dreptunghiului sursă inversat, cu culorile dreptunghiului destinaţie, folosind operatorul boolean OR (SAU).
NOTSRCCOPY	Copiază dreptunghiul sursă inversat, în destinaţie.
NOTSRCERASE	Combină culorile dreptunghiurilor sursă şi destinaţie prin utilizarea operatorului boolean OR (SAU) şi apoi inversează culoarea rezultată.
PATCOPY	Copiază modelul specificat în blocul bitmap destinaţie.
PATINVERT	Combină culorile modelului cu culorile dreptunghiului de destinaţie, folosind operatorul boolean XOR.
PATPAINT	Combină culorile modelului cu culorile dreptunghiului sursă inversat, folosind operatorul boolean OR. Rezultatul acestei operaţii este combinat cu culorile dreptunghiului destinaţie folosind operatorul boolean OR.
SRCAND	Combină culorile dreptunghiurilor sursă şi destinaţie folosind operatorul boolean AND.
SRCCOPY	Copiază dreptunghiul sursă direct în dreptunghiul destinaţie.
SRCERASE	Combină culorile inversate ale dreptunghiului destinaţie cu culorile dreptunghiului sursă, folosind operatorul boolean AND.

Valoare	Descriere
SRCINVERT	Combină culorile din dreptunghiul sursă cu cele din dreptunghiul destinație folosind operatorul boolean XOR.
SRCPAINT	Combină culorile din dreptunghiul sursă cu cele din dreptunghiul destinație folosind operatorul boolean OR.
WHITENESS	Umple dreptunghiul destinație folosind culoarea asociată cu indicele 1 din paleta fizică. (Această culoare este alb în paleta fizică implicită.)

Tabelul 1435.2 *Codurile de operații cu rastru obișnuite, aplicate blocurilor bitmap.*

Dacă o rotație sau o așchiere (o transformare care modifică lungimea aparentă și orientarea liniilor verticale sau orizontale dintr-un obiect) este în execuție în contextul de dispozitiv sursă, *BitBlt* returnează o eroare. Dacă există alte transformări în contextul de dispozitiv sursă (iar o transformare de corespondență nu este în execuție în contextul de dispozitiv destinație), dreptunghiul din contextul de dispozitiv destinație este extins, comprimat sau rotit, după cum este necesar.

Dacă formatele color ale contextelor de dispozitiv sursă și destinație nu se potrivesc, funcția *BitBlt* convertește formatul color al sursei la formatul color al destinației. Când metafișierul extins este în curs de înregistrare, va apărea o eroare când contextul de dispozitiv sursă identifică un context de dispozitiv de metafișier extins. *BitBlt* va returna o eroare când contextele sursă și destinație reprezintă dispozitive diferite (asta înseamnă că ar trebui să creați întotdeauna contextul sursă cu funcția *CreateCompatibleDC*).

Pentru a înțelege mai bine prelucrările efectuate de funcția *BitBlt*, analizați programul *Happy_Face.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Happy_Face.cpp* încarcă un bitmap în memorie, apoi îl așează în mozaic în zona client a ferestrei. Prelucrarea operativă se află în funcția *WndProc*.

PRODUCEREA BLOCURILOR BITMAP DIB — C/C++ 1436

Windows acceptă atât blocuri bitmap dependente de dispozitiv, cât și independente de dispozitiv. În secțiunea 1435 de exemplu, ați produs un bitmap dependent de dispozitiv, apoi l-ați afișat pe ecran. Pentru a afișa blocuri bitmap independente de dispozitiv (DIB), programele dumneavoastră trebuie să efectueze prelucrări asemănătoare. Înainte de a afișa un bitmap independent de dispozitiv, trebuie să îl convertiți într-unul dependent de dispozitiv. În acest scop, veți utiliza funcția *CreateDIBitmap* care produce un bitmap dependent de dispozitiv dintr-unul independent de dispozitiv și, opțional, stabilește biții de bitmap. Veți implementa funcția *CreateDIBitmap* potrivit prototipului prezentat mai jos:

```
HBITMAP CreateDIBitmap(
    HDC hdc,      // identificator pentru contextul de dispozitiv
    CONST BITMAPINFOHEADER *lpbmih, // pointer la dimensiunea
                                    // blocului bitmap si la
                                    // datele de format
    DWORD fdwInit,   // indicatoare de initializare
    CONST VOID *lpbInit, // pointer la datele de initializare
    CONST BITMAPINFO *lpbmi, // pointer la datele de format color
    UINT fuUsage     // utilizare date color
);
```

Funcția *CreateDIBitmap* acceptă parametrii prezentați în Tabelul 1436.1.

Parametru	Descriere
hdc	Identifică un context de dispozitiv.
lpbmib	Indică o structură *BITMAPINFOHEADER*. Dacă *fdwInit* are valoarea *CBM_INIT*, funcția utilizează structura *BITMAPINFOHEADER* pentru a obține lățimea și înălțimea dorite ale blocului bitmap precum și alte informații. Remarcați că o valoare pozitivă pentru înălțime indică un DIB de tip jos-sus, iar una negativă indică un DIB de tip sus-jos. Acest scenariu este compatibil cu funcția *CreateDIBitmap*.
fdwInit	Un grup de indicatoare pe biți care specifică modul în care sistemul de operare inițializează biții blocului bitmap. Programele pot specifica doar o constantă pentru *fdwInit*. Valoarea parametrului trebuie să fie *CBM_INIT* sau zero. Dacă acest indicator este activat, sistemul de operare va folosi datele indicate de *lpbInit* și de *lpbmi* pentru a inițializa biții din bitmap. Dacă acest parametru este zero, funcția nu va folosi datele indicate de acei parametri. Dacă *fdwInit* este zero, sistemul de operare nu va inițializa biții din bitmap.
lpbInit	Indică o matrice de biți care conține datele inițiale ale blocului bitmap. Formatul datelor depinde de membrul *biBitCount* din structura *BITMAPINFO* indicată de parametrul *lpbmi*.
lpbmi	Indică o structură *BITMAPINFO* care descrie dimensiunile și formatul de culoare ale matricei indicate de parametrul *lpbInit*.
fuUsage	Specifică dacă a fost inițializat membrul *bmiColors* din structura *BITMAPINFO* și, în caz că a fost, dacă *bmiColors* conține explicit indicii de valoare sau de paletă pentru roșu, verde și albastru (RGB). Parametrul *fuUsage* trebuie să conțină una din valorile din Tabelul 1436.2.

***Tabelul 1436.1** Parametrii funcției **CreateDIBitmap**.*

Parametrul *fuUsage* trebuie să fie una din cele două constante descrise în Tabelul 1436.2.

Valoare	Semnificație
DIB_PAL_COLORS	Fișierul bitmap conține o tabelă de culoare constând într-o matrice de indici pe 16 biți din cadrul paletei logice a contextului de dispozitiv din care va fi selectat blocul bitmap.
DIB_RGB_COLORS	Fișierul bitmap conține o tabelă de culori care conține valorile literale RGB.

***Tabelul 1436.2** Valorile posibile ale parametrului **fuUsage**.*

Dacă funcția reușește, valoarea returnată va fi un identificator de bitmap. Dacă funcția eșuează, valoarea de returnare va fi NULL. Dacă nu mai este nevoie de bitmap, apelați funcția *DeleteObject* pentru a-l șterge.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateDIBitmap*, analizați programul *Create_DIB_Bitmap.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Create_DIB_Bitmap.cpp* încarcă un bitmap independent de dispozitiv, desenează blocul bitmap într-un context de dispozitiv de memorie, apoi transferă contextul de dispozitiv de memorie într-un context de dispozitiv de ecran (fereastra programului). Ca de obicei, prelucrarea operativă are loc în funcția *WndProc*.

UMPLEREA CU UN MODEL A UNUI DREPTUNGHI

C/C++ 1437

Programele dumneavoastră pot utiliza blocuri bitmap şi adăuga informaţii grafice în zona de afişare a ferestrei. În plus, ele pot utiliza creioane (peniţe) şi contexte de dispozitiv de memorie pentru a desena forme simple sau complexe în fereastră. De exemplu, programul dumneavoastră poate utiliza funcţia *PatBlt* pentru desenarea dreptunghiurilor într-un context de dispozitiv. Funcţia *PatBlt* trasează dreptunghiul dat folosind pensula selectată la momentul curent în contextul de dispozitiv specificat. Culoarea pensulei şi culorile de suprafaţă sunt combinate utilizând operaţia cu rastru dată. Veţi implementa funcţia *PatBlt* potrivit prototipului prezentat mai jos:

```
BOOL PatBlt(
   HDC hdc,        // identificator pentru contextul de dispozitiv
   int nXLeft,     // coord. x a coltului stanga sus a
                   // dreptunghiului de umplut
   int nYLeft,     // coord. y a coltului stanga sus a
                   // dreptunghiului de umplut
   int nWidth,     // latimea dreptunghiului de umplut
   int nHeight,    // inaltimea dreptunghiului de umplut
   DWORD dwRop     // coduri de operatii cu rastru
);
```

Funcţia *PatBlt* acceptă parametrii prezentaţi în Tabelul 1437.1.

Parametru	Descriere
hdc	Identifică contextul de dispozitiv.
nXLeft	Conţine coordonata x, în unităţi logice, a colţului din stânga sus al dreptunghiului de umplut.
nYLeft	Conţine coordonata y, în unităţi logice, a colţului din stânga sus al dreptunghiului de umplut.
nWidth	Specifică lăţimea, în unităţi logice, a dreptunghiului de umplut.
nHeight	Specifică înălţimea, în unităţi logice, a dreptunghiului de umplut.
dwRop	Conţine codul operaţiei cu rastru. Acest cod este una din valorile descrise în Tabelul 1437.2.

Tabelul 1437.1 *Parametrii funcţiei* ***PatBlt***.

Aşa cum aţi văzut în Tabelul 1437.1, parametrul *dwRop* acceptă o serie de coduri de operaţii cu rastru. Tabelul 1437.2 enumeră codurile operaţiilor cu rastru.

Valoare	Semnificaţie
PATCOPY	Copiază modelul specificat în blocul bitmap de destinaţie.
PATINVERT	Combină culorile modelului specificat cu culorile dreptunghiului destinaţie, folosind operatorul boolean XOR.
DSTINVERT	Inversează dreptunghiul de destinaţie.
BLACKNESS	Umple dreptunghiul destinaţie folosind culoarea asociată cu indicele 0 din paleta fizică. (Această culoare este negru în paleta fizică implicită.)

(continuare)

Valoare	Semnificație
WHITENESS	Umple dreptunghiul destinație folosind culoarea asociată cu indicele 1 din paleta fizică. (Această culoare este alb în paleta fizică implicită.)

Tabelul 1437.2 Codurile operațiilor cu rastru.

Valorile parametrului *dwRop* pentru această funcție sunt un subgrup limitat la 256 de coduri de operații cu rastru ternare; în particular, un cod de operație care se referă la un dreptunghi sursă nu poate fi utilizat. Pentru a afla lista completă a operațiilor cu rastru ternare, consultați documentația *on-line* a compilatorului dumneavoastră. Nu toate dispozitivele acceptă funcția *PatBlt*. E bine ca înaintea invocării funcției *PatBlt* pentru dispozitiv să utilizați funcția *GetDeviceCaps* pentru a afla caracteristica *RC_BITBLT* a dispozitivului.

Pentru a înțelege mai bine prelucrările făcute de funcția *PatBlt*, analizați programul *Paint_Rect.cpp*, de pe CD-ROM-ul atrașat. Acest program trasează o casetă gri cu un chenar tridimensional în fereastra programului. El utilizează trei pensule standard, pentru a trasa dreptunghiurile care alcătuiesc chenarul tridimensional al casetei. Prelucrarea operativă se face în funcția *WndProc*.

1438 UTILIZAREA FUNCȚIEI SETDIBITS



Pe măsură ce programele dumneavoastră devin mai complexe, vor exista situații în care să fie nevoie să schimbați schema de culoare a unui bitmap. Bitmapurile independente de dispozitiv mențin informații de culoare în antetul de bitmap. Puteți utiliza culorile stocate anterior de programul de creare în antetul unui bitmap independent de dispozitiv pentru schimbarea culorilor într-un bitmap dat. Funcția *SetDIBits* stabilește pixelii într-un bitmap folosind datele de culoare găsite în blocul bitmap independent de dispozitiv specificat. Veți implementa funcția *SetDIBits* potrivit prototipului prezentat mai jos:

```
int SetDIBits(
    HDC hdc,                      // identificator pentru contextul
                                  // diapozitiv
    HBITMAP hbmp,                 // identificator pentru bitmap
    UINT uStartScan,              // linia de start la scanare
    UINT cScanLines,              // numar linii de scanare
    CONST VOID *lpvBits,          // matrice cu bitii de bitmap
    CONST BITMAPINFO *lpbmi,      // adresa structurii cu datele de
                                  // bitmap
    UINT fuColorUse               // tip de indici de culoare
                                  // utilizati
);
```

Funcția *SetDIBits* acceptă parametrii descriși în Tabelul 1438.1.

Parametru	Descriere
hdc	Identifică contextul de dispozitiv.
hbmp	Identifică blocul bitmap DIB care va fi modificat folosind datele de culoare de la blocul bitmap DIB specificat.

Parametru	Descriere
uStartScan	Specifică începutul liniei de scanare pentru datele de culoare independente de dispozitiv din matricea indicată de parametrul *lpvBits*.
cScanLines	Specifică numărul de linii de scanare găsite în matricea care conţine datele de culoare independente de dispozitiv.
lpvBits	Indică datele de culoare ale blocului DIB, stocate sub forma unei matrice de octeţi. Formatul valorilor din bitmap depinde de membrul *biBitCount* din structura *BITMAPINFO* indicată de parametrul *lpbmi*.
lpbmi	Indică o structură de date *BITMAPINFO* care conţine informaţii despre DIB.
fuColorUse	Specifică dacă a fost furnizat membrul *bmiColors* al structurii *BITMAPINFO* şi, în caz afirmativ, dacă *bmiColors* conţine explicit indicii de valoare sau de paletă ai culorilor roşu, verde şi albastru (RGB). Parametrul trebuie să ia una din valorile din Tabelul 1438.2.

Tabelul 1438.1 *Parametrii acceptaţi de funcţia* ***SetDIBits***.

Conform Tabelului 1438.1, parametrul *fuColorUse* acceptă o serie de valori constante prestabilite enumerate în Tabelul 1438.2 împreună cu semnificaţia lor.

Valoare	Semnificaţie
DIB_PAL_COLORS	Tabela de culoare constă într-o matrice de indici pe 16 biţi din cadrul paletei logice a contextului de dispozitiv identificat de parametrul hdc.
DIB_RGB_COLORS	Blocul bitmap conţine o tabelă de culori care conţine valorile literale RGB.

Tabelul 1438.2 *Valorile acceptate de parametrul* ***fuColorUse***.

Dacă funcţia *SetDIBits* reuşeşte, valoarea de returnare este numărul de linii de scanare copiate cu succes de *SetDIBits*. Dacă eşuează, valoarea de returnare va fi zero. Windows atinge viteza optimă de desenare a blocului bitmap când biţii din bitmap sunt indici în paleta sistemului.

Aplicaţiile pot regăsi paleta şi indicii de culoare ai sistemului prin apelarea funcţiei *GetSystemPaletteEntries*. După ce culorile şi indicii au fost regăsiţi, aplicaţia poate produce blocul blocul bitmap independent de dispozitiv. Contextul de dispozitiv identificat de parametrul *hdc* este utilizat numai când parametrul *fuColorUse* are ca valoare constanta *DIB_PAL_COLORS*, altfel este ignorat. Programele nu trebuie să fi selectat într-un context de dispozitiv blocul bitmap identificat de parametrul *hbmpi* când aplicaţia apelează funcţia *GetSystemPaletteEntries*. Originea blocurilor DIB de tip jos-sus este colţul din stânga jos al blocului bitmap, iar a celor sus-jos este colţul stânga sus al blocului bitmap.

Pentru a înţelege mai bine prelucrările efectuate de funcţia *SetDIBits*, analizaţi programul *Change_Colors.cpp* conţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *Change_Colors.cpp* trasează un bitmap independent de dispozitiv în zona client a ferestrei. Iniţial, programul aplică blocului bitmap culorile alb şi negru. Când utilizatorul selectează opţiunea *Test!* din meniu, programul modifică datele din bitmap, astfel încât fiecare succesiune de doi pixeli negri să reflecte combinaţia de pixeli albastru-negru. Prelucrarea operativă are loc în funcţia *WndProc*.

1439 SETDIBITSTODEVICE UTILIZATĂ PENTRU AFIȘAREA UNUI BITMAP PE UN DISPOZITIV DAT

C/C++

Așa cum ați învățat în secțiunea 1438, programele dumneavoastră pot utiliza funcția *SetDIBits* pentru stabilirea biților de culoare ai unui bitmap în concordanță cu valorile conținute de antetul unui bitmap independent de dispozitiv dat. În mod frecvent, programele trebuie să reducă numărul de culori ale unui bitmap independent de dispozitiv pentru un dispozitiv dat. De exemplu, Windows trebuie să mapeze un bitmap de 256 de culori la unul de 20 de culori, în cazul în care încercați să afișați acel bloc bitmap pe un monitor VGA. Funcția *SetDIBitsToDevice* trasează un bitmap independent de dispozitiv pe dispozitivul asociat cu contextul de dispozitiv dat. Funcția *SetDIBitsToDevice* stabilește pixelii din blocul bitmap desenat, folosind datele de culoare din blocul bitmap independent de dispozitiv. Veți implementa funcția *SetDIBitsToDevice* potrivit prototipului prezentat mai jos:

```
int SetDIBitsToDevice(
    HDC hdc,    // identificator pentru contextul de dispozitiv
    int XDest,  // coordonata x a coltului din stanga sus al
                // drept. destinatie
    int YDest,  // coordonata y a coltului din stanga sus al
                // drept. destinatie
    DWORD dwWidth, // latimea dreptunghiului sursa
    DWORD dwHeight, // inaltimea dreptunghiului sursa
    int XSrc, // coordonata x a coltului din stanga jos al
              // drept. sursa
    int YSrc, // coordonata y a coltului din stanga jos al
              // drept. sursa
    UINT uStartScan, // prima linie de scanare din matrice
    UINT cScanLines, // numar linii de scanare
    CONST VOID *lpvBits, // adresa matricei cu bitii din DIB
    CONST BITMAPINFO *lpbmi, // adresa structurii cu informatii
                             // despre bitmap
    UINT fuColorUse // RGB sau indicii de paleta
);
```

Funcția *SetDIBitsToDevice* acceptă parametrii prezentați în Tabelul 1439.1.

Parametru	Descriere
hdc	Identifică contextul de dispozitiv.
XDest	Conține coordonata x, în unități logice, a colțului din stânga sus al dreptunghiului destinație.
YDest	Conține coordonata y, în unități logice, a colțului din stânga sus al dreptunghiului destinație.
dwWidth	Specifică lățimea, în unități logice, a blocului bitmap independent de dispozitiv.
dwHeight	Specifică înălțimea, în unități logice, a blocului bitmap independent de dispozitiv.

Parametru	Descriere
XSrc	Conține coordonata x, în unități logice, a colțului din stânga jos al blocului bitmap independent de dispozitiv.
YSrc	Conține coordonata y, în unități logice, a colțului din stânga jos al blocului bitmap independent de dispozitiv.
uStartScan	Specifică prima linie de scanare dintr-un bitmap independent de dispozitiv.
cScanLines	Specifică numărul de linii de scanare ale blocului bitmap independent de dispozitiv conținute în matricea indicată de parametrul *lpvBits*.
lpvBits	Indică datele despre culoarea unui bitmap independent de dispozitiv, stocate de programul care l-a creat sub forma unei matrice de biți.
lpbmi	Indică o structură *BITMAPINFO* care conține informații despre blocul bitmap independent de dispozitiv.
fuColorUse	Indică dacă membrul *bmiColors* al structurii *BITMAPINFO* conține explicit valorile sau indicii dintr-o paletă pentru culorile roșu, verde și albastru (RGB). Parametrul trebuie să ia una din valorile din Tabelul 1438.2

***Tabelul 1439** Parametrii acceptați de funcția **SetDIBitsToDevice**.*

Așa cum ați aflat din Tabelul 1439.1 parametrul *fuColorUse* va accepta una din cele două valori predefinite. Tabelul 1438.2 enumeră valorile acceptate pentru parametrul *fuColorUse* în cazul funcției *SetDIBitsToDevice*.

Funcția va avea o viteză optimă de desenare a blocului bitmap atunci când biții lui sunt indici în paleta sistemului. Aplicațiile pot apela funcția *GetSystemPaletteEntries* pentru regăsirea indicilor și culorilor paletei sistem. După ce aplicația a regăsit culorile și indicii, ea poate produce blocul bitmap independent de dispozitiv.

Originea blocului bitmap independent de dispozitiv de tip jos-sus este colțul din stânga jos al blocului, iar originea celui sus-jos este colțul din stânga sus. Pentru a reduce cantitatea de memorie cerută pentru stabilirea biților într-un bitmap independent de dispozitiv mare pe o suprafață de dispozitiv, aplicația poate grupa ieșirea în benzi prin apelarea repetată a funcției *SetDIBitsToDevice*, plasând de fiecare dată o porțiune diferită de bitmap în matricea *lpvBits*. Valorile parametrilor *uStartScan* și *cScanLines* identifică acea parte de bitmap conținută în matricea *lpvBits*. Funcția *SetDIBitsToDevice* returnează o eroare dacă este apelată de un proces care rulează în fundal în timp ce o sesiune MS-DOS extinsă pe întregul ecran rulează în prim plan.

Pentru a înțelege mai bine prelucrările efectuate de funcția *SetDIBitsToDevice*, analizați programul *Draw_2_Boxes.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Draw_2_Boxes.cpp* utilizează un bitmap independent de dispozitiv și o paletă personalizată. Programul trasează blocul bitmap independent de dispozitiv la dimensiunea sa normală prin intermediul funcției *SetDIBitsToDevice* apoi utilizează funcția *StretchDIBits* pentru a afișa blocul bitmap extins la un procent de 200% din dimensiunea sa normală. Prelucrarea operativă are loc în programele handler pentru mesajele *WM_CREATE* și *WM_PAINT* din cadrul funcției *WndProc*.

METAFIȘIERELE

C/C++ 1440

În secțiunile precedente ați învățat despre blocuri bitmap. Ați aflat, de asemenea, că Windows acceptă trei tipuri grafice de bază. Al doilea tip este cunoscut sub denumirea de

metafișier. Metafișierele conțin secvențe codificate de apeluri de funcții de interfață cu dispozitive grafice. Când programul *execută* un metafișier, rezultatul este ca și atunci când programul ar utiliza direct funcția de interfață grafică. Puteți considera metafișierul ca un macro pentru grafică. Puteți stoca metafișierele în memorie (sau ca fișiere pe disc), puteți reîncărca metafișierul și îl puteți executa în oricât de multe aplicații. În plus, metafișierele sunt mai independente de dispozitiv decât blocurile bitmap, deoarece calculatorul (și dispozitivul) vor interpreta funcția de interfața cu dispozitivul grafic pe baza contextului de dispozitiv de ieșire.

Windows 95 și Windows NT asigură suport pentru metafișiere Windows 3.x. Însă, sistemul de operare Win32 asigură suport pentru noul tip de *metafișier extins*. Programele dumneavoastră ar trebui să folosească acest nou tip de metafișier. Cele mai importante diferențe dintre metafișierul Windows 3.x și metafișierul extins constau în aceea că metafișierele extinse sunt cu adevărat independente de dispozitiv și prezintă suport pentru noile funcții Win32 GDI API.

Este important să înțelegem că interfața Win32 API acceptă în întregime tipul metafișier din Win16. De fapt, interfața Win32 API conține două seturi de funcții metafișier. Un set pentru interfața Win16 API și un altul pentru interfața Win32 API. De exemplu, programele dumneavoastră pot apela funcția *PlayMetaFile* pentru executarea unui metafișier pe 16 biți. În aceeași manieră, programul poate apela funcția *PlayEnhMetaFile* pentru executarea unui metafișier pe 32 de biți. Pentru claritate, secțiunile care urmează se concentrează pe metafișierele extinse și mai puțin pe metafișierele Win16.

1441 CREAREA ȘI AFIȘAREA UNUI METAFIȘIER C/C++

Așa cum ați învățat în secțiunea 1440, metafișierele sunt o serie de instrucțiuni de interfață cu dispozitivele grafice stocate anterior de programul care l-a creat într-o structură care poate fi salvată. Veți utiliza funcțiile interfeței pentru dispozitivele grafice într-un context metafișier de dispozitiv pentru a crea metafișiere. Se va utiliza un *context de dispozitiv de referință*, oferind metafișierului un suport pentru menținerea dimensiunilor la dispozitivele de ieșire. Dispozitivul de referință corespunde cu dispozitivul pe care a apărut inițial imaginea grafică. Pentru producerea metafișierelor se va utiliza funcția *CreateEnhMetaFile* care creează un context de dispozitiv pentru un format extins de metafișier. Contextul de dispozitiv creat poate fi utilizat pentru stocarea unei imagini independente de dispozitiv. Veți implementa funcția *CreateEnhMetaFile* potrivit prototipului prezentat mai jos:

```
HDC CreateEnhMetaFile(
    HDC hdcRef,           // identificator pentru un context de
                          // dispozitiv de referinta
    LPCTSTR lpFilename,   // pointer la un sir nume fisier
    CONST RECT *lpRect,   // pointer la un dreptunghi de
                          // incadrare
    LPCTSTR lpDescription // pointer la un sir optional de
                          // descriere
);
```

Funcția *CreateEnhMetaFile* acceptă parametrii prezentați în Tabelul 1441.

Parametru	Descriere
hdcRef	Identifică un dispozitiv de referință pentru un metafișier extins.
lpFilename	Indică numele fișierului în care va fi produs metafișierul extins. Dacă acest parametru este NULL, metafișierul extins se bazează pe memorie, iar conținutul său se pierde când este șters cu funcția *DeleteEnhMetaFile*.
lpRect	Indică o structură *RECT* care specifică dimensiunile (în unități de 0,01 milimetri) imaginii ce va fi stocată în final în metafișierul extins.
lpDescription	Indică un șir care specifică numele aplicației care a creat imaginea și titlul său. Șirul indicat de *lpDescription* trebuie să conțină un caracter NULL între numele aplicației și numele imaginii și trebuie să se termine cu două caractere NULL, de exemplu, „Editor grafic XYZ\0Peisaj marin\0\0", unde \0 reprezintă caracterul NULL. Dacă *lpDescription* este NULL, nu există nici o intrare corespunzătoare în antetul metafișierului extins.

***Tabelul 1441** Parametrii funcției **CreateEnhMetaFile**.*

Windows utilizează dispozitivul de referință identificat de parametrul *hdcRef* pentru înregistrarea rezoluției și unităților dispozitivului pe care a apărut inițial imaginea. Dacă parametrul *hdcRef* este NULL, el folosește dispozitivul curent de afișare ca referință.

Membrii *left* și *top* ai structurii *RECT* indicată de parametrul *lpRect* trebuie să fie mai mici decât membrii *right* și respectiv *bottom* ai aceleiași structuri. Sunt incluse în imaginea grafică și punctele de pe marginea dreptunghiului. Dacă *lpRect* este NULL, interfața cu dispozitivul grafic (GDI) calculează dimensiunile celui mai redus dreptunghi care înconjoară imaginea desenată de aplicație. Parametrul *lpRect* trebuie furnizat oricând este posibil.

Aplicațiile utilizează contextul de dispozitiv creat de funcția *CreatEnh.MetaFile* pentru a stoca o imagine grafică într-un metafișier extins. Identificatorul care identifică acest context de dispozitiv poate fi transmis oricărei funcții GDI.

După ce o aplicație a stocat imaginea într-un metafișier extins, ea poate afișa imaginea pe orice dispozitiv de ieșire prin apelarea funcției *PlayEnhMetaFile*. La afișarea imaginii, Windows utilizează dreptunghiul punctat de parametrul *lpRect* și datele de rezoluție din dispozitivul de referință la poziția și scara imaginii grafice. Contextul de dispozitiv returnat de această funcție conține aceleași atribute implicite asociate cu orice context de dispozitiv nou. Aplicațiile trebuie să utilizeze funcția *GetWinMetaFileBits* pentru a converti un metafișier extins la un format de metafișier Windows mai vechi. Numele fișierului pentru metafișierul extins trebuie să folosească extensia .EMF.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateEnhMetaFile*, analizați programul *Crosshatch_Box.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Crosshatch_Box.cpp* produce un metafișier extins al unui dreptunghi umplut la pornire cu hașuri. Programul apoi execută metafișierul în rutina de prelucrare a mesajului *WM_PAINT*. Programul scalează metafișierul luând ca bază zona client a ferestrei – ceea ce înseamnă că dimensiunea dreptunghiului se va schimba o dată cu redimensionarea ferestrei. Prelucrarea operativă are loc în programele handler pentru mesajele *WM_CREATE* și *WM_PAINT*.

1442 ENUMERAREA METAFIȘIERELOR EXTINSE



Programul dumneavoastră poate folosi metafișiere pentru a reutiliza instrucțiuni stocate în interfața cu dispozitivul grafic. Puteți stoca mai multe seturi de instrucțiuni într-un metafișier dat. Fiecare set de instrucțiuni este recunoscut drept o înregistrare în metafișier. Mai târziu, când accesați metafișierul, puteți să accesați un set specific de instrucțiuni sau să aflați ce instrucțiuni sunt în metafișier. Pentru aceasta, programul poate apela funcția *EnumEnhMetaFile* pentru a enumera toate înregistrările dintr-un metafișier extins prin regăsirea fiecărei înregistrări și transmiterea ei către funcția *callback* specificată. Funcția *callback* furnizată de aplicație prelucrează fiecare înregistrare după cum e cazul. Enumerarea continuă până când ultima înregistrare este prelucrată sau când funcția callback returnează zero. Veți implementa funcția *EnumEnhMetaFile* potrivit prototipului prezentat mai jos:

```
BOOL EnumEnhMetaFile(
    HDC hdc,     // identificator pentru contextul de dispozitiv
    HENHMETAFILE hemf, // identificator pentru metafisierul extins
    ENHMFENUMPROC lpEnhMetaFunc, // pointer la functia callback
    LPVOID lpData,    // pointer la datele functiei callback
    CONST RECT *lpRect // pointer la dreptunghiul de incadrare
);
```

Funcția *EnumEnhMetaFile* acceptă parametrii descriși în Tabelul 1442.1.

Parametru	Descriere
hdc	Identifică un context de dispozitiv. Acest identificator este transmis funcției callback.
hemf	Identifică metafișierul extins.
lpEnhMetaFunc	Indică funcția callback furnizată de aplicație.
lpData	Indică datele opționale ale funcției callback.
lpRect	Indică o structură *RECT* care conține coordonatele colțurilor din stânga sus și dreapta jos ale imaginii. Dimensiunile acestui dreptunghi sunt specificate în unități logice.

Tabelul 1442.1 *Parametrii acceptați de funcția* ***EnumEnhMetaFile***.

Așa cum știți, o funcție de enumerare API utilizează o funcție callback pentru a prelucra informațiile returnate de funcția de enumerare. Funcția *EnumEnhMetaFile* utilizează o funcție *callback* cu formatul generalizat al funcției *EnhMetaFileProc*. Funcția *EnhMetaFileProc* prelucrează înregistrările formatelor extinse de metafișiere. Veți implementa funcția *EnhMetaFileProc* potrivit prototipului prezentat mai jos:

```
int CALLBACK EnhMetaFileProc(
    HDC hDC, // identificator pentru contextul de dispozitiv
    HANDLETABLE FAR *lpHTable, // pointer la tabela
                               // identificatori ai
                               // metafisierului
    ENHMETARECORD FAR *lpEMFR, // pointer la inregistrarea de
                               // metafisier
    int nObj, // numar de obiecte
```

```
    LPARAM lpData // pointer la date optionale
);
```

Funcţia *EnhMetaFileProc* acceptă parametrii descrişi în Tabelul 1442.2.

Parametru	Descriere
hDC	Identifică contextul de dispozitiv transmis funcţiei *EnumEnhMetaFile*.
lpHTable	Indică o tabelă de identificatori asociaţi cu obiectele grafice (creioane, pensule etc.) din metafişier. Prima intrare conţine identificatorul metafişierului extins.
lpEMFR	Indică una din înregistrările metafişierului. Această înregistrare nu trebuie să fie modificată. (Dacă sunt necesare modificări, e bine să fie efectuate pe o copie a înregistrării.)
nObj	Specifică numărul de obiecte cu identificatorii asociaţi în tabela de identificatori.
lpData	Indică orice date furnizate de aplicaţie.

***Tabelul 1442.2** Parametrii acceptaţi de funcţia **EnhMetaFileProc**.*

O aplicaţie trebuie să îşi transmită adresa către funcţia *EnumEnhMetaFile* pentru a înregistra funcţia callback. Ca şi în cazul celorlalte funcţii callback despre care aţi învăţat în secţiunile anterioare, funcţia *EnhMetaFileProc* este un substituent pentru numele funcţiei definite de aplicaţie.

Punctele de pe marginea dreptunghiului indicate de parametrul *lpRect* sunt incluse în imagine. Dacă parametrul *hdc* este NULL, Windows ignoră parametrul *lpRect*. Dacă funcţia callback apelează funcţia *PlayEnhMetaFileRecord*, parametrul *hdc* trebuie să identifice un context de dispozitiv valid. Pentru a transforma imaginea afişată prin intermediul funcţiei *PlayEnhMetaFileRecord*, Windows utilizează modurile de transformare şi de mapare ale contextului de dispozitiv. Puteţi utiliza funcţia *EnumEnhMetaFile* pentru a include un metafişier extins într-un alt metafişier.

Pentru a înţelege mai bine prelucrările efectuate de funcţia *EnumEnhMetaFile*, analizaţi programul *Draw_Shapes.cpp* conţinut pe CD-ROM-ul care însoţeşte cartea de faţă. Programul *Draw_Shapes.cpp* încarcă un metafişier extins şi îl afişează în zona client a ferestrei. Când utilizatorul selectează opţiunea *Test!* din meniu, programul execută din nou metafişierul, de această dată folosind funcţia *EnumEnhMetaFile*. Funcţia callbacl interceptează înregistrarea *EMR_CREATEBRUSHINDIRECT* din metafişier şi o înlocuieşte cu o pensulă gri închis. Prelucrarea operativă are loc în procedurile *WndProc* şi *PaintMetaFile*. Când compilaţi şi executaţi programul *Draw_Shapes.cpp*, el mai întâi desenează formele cu haşuri, apoi le redesenează într-un gri plin.

UTILIZAREA FUNCŢIEI GETWINMETAFILEBITS

C/C++ 1443

Programele Win32 ar trebui să folosească structurile de metafişiere extinse. Însă, dacă programele dumneavoastră vor rula pe mai multe platforme, se vor ivi situaţii când programele dumneavoastră vor trebui să convertească metafişiere extinse în metafişiere de tip Windows 3.x. Funcţia *GetWinMetaFileBits* converteşte înregistrările în format extins ale unui metafişier la formatul Windows şi stochează înregistrările convertite într-un buffer

specificat. Veți implementa funcția *GetWinMetaFileBits* potrivit prototipului prezentat mai jos:

```
UINT GetWinMetaFileBits(
    HENHMETAFILE hemf,  // identificator pentru metafisierul
                        // extins
    UINT cbBuffer,      // marime buffer
    LPBYTE lpbBuffer,   // pointer la buffer
    INT fnMapMode,      // mod de mapare
    HDC hdcRef          // identificator pentru contextul de
                        // dispozitiv de referinta
);
```

Funcția *GetWinMetaFileBits* acceptă parametrii prezentați în Tabelul 1443.

Parametru	Descriere
hemf	Identifică metafișierul extins.
cbBuffer	Specifică mărimea, în octeți a bufferului în care vor fi copiate înregistrările convertite.
lpbBuffer	Indică bufferul în care vor fi copiate înregistrările convertite. Dacă *lpbBuffer* este NULL, *GetWinMetaFileBits* returnează numărul de octeți necesari pentru stocarea înregistrărilor convertite ale metafișierului.
fnMapMode	Specifică modul de mapare ce trebuie utilizat de programele dumneavoastră cu metafișierul convertit.
hdcRef	Identifică contextul de dispozitiv de referință.

Tabelul 1443 *Parametrii funcției* ***GetWinMetaFileBits***.

Dacă funcția reușește, iar pointerul de buffer este NULL, valoarea de returnare este numărul de octeți necesari pentru stocarea înregistrărilor convertite. Dacă funcția reușește, iar pointerul la buffer este un pointer valid, valoarea de returnare este mărimea datelor metafișierului, exprimată în octeți. Dacă funcția eșuează, valoare de returnare este NULL.

Funcția *GetWinMetaFileBits* convertește un metafișier extins într-un metafișier de format Windows, astfel încât el să poată fi afișat într-o aplicație care recunoaște vechiul format. Windows folosește contextul de dispozitiv de referință pentru determinarea rezoluției metafișierului convertit. Funcția *GetWinMetaFileBits* nu invalidează identificatorul de metafișier extins. Aplicațiile trebuie să apeleze funcția *DeleteEnhMetaFile* pentru eliberarea identificatorului când nu mai este nevoie de el.

Datorită limitării metafișierului de format Windows, unele informații pot fi pierdute din conținutul metafișierului regăsit. De exemplu, un apel inițial la funcția *PolyBezier* în metafișierul extins poate fi convertit într-un apel la funcția *Polyline* din metafișierul de format Windows, deoarece nu există o funcție echivalentă *PolyBezier* în formatul Windows.

Aplicațiile Windows 3.x definesc originea și extensia portului de vizualizare al unei imagini stocate într-un metafișier de format Windows. Ca rezultat, înregistrările în format Windows produse de *GetWinMetaFileBits* nu conțin funcțiile *SetViewportOrgEx* și *SetViewportExtEx*. Însă funcția *GetWinMetaFileBits* produce înregistrări de format Windows pentru funcțiile *SetWindowExtEx* și *SetMapMode*. Pentru producerea unui metafișier de format Windows scalabil, specificați valoarea *MM_ANISOTROPIC* pentru parametrul *fnMapMode*. Windows

întotdeauna mapează colțul din stânga sus al imaginii metafișierului la originea dispozitivului de referință.

Pentru a înțelege mai bine prelucrările efectuate de funcția *GetWinMetaFileBits*, analizați programul *Convert_EMF_WMF.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Convert_EMF_WMF.cpp* convertește un metafișier extins dat (în cazul nostru *sample.emf*) la un metafișier Windows 3.x standard. Programul apoi salvează metafișierul convertit ca *sample.wmf*. Prelucrarea operativă a programului are loc în rutina de prelucrare a mesajului *WM_COMMAND*.

PICTOGRAMELE

C/C++ 1444

Windows acceptă trei tipuri de bază de fișiere grafice. Programele vor utiliza tipurile bitmap și metafișier pentru gestionarea imaginilor grafice mari sau mici din cadrul zonei client a ferestrei și chiar în fereastra însăși. *Pictograma* este o subclasă a tipului bitmap. Însă, setul de activități pe care îl veți efectua cu pictogramele este suficient de delimitat pentru ca Windows să trateze pictogramele ca un tip separat de fișier grafic.

Pictogramele sunt blocuri bitmap de mici dimensiuni pe care Windows le utilizează ca reprezentări ale unor obiecte cum ar fi aplicații, fișiere și directoare. În Windows 95, veți vedea pictograme în fiecare aspect al interfeței cu utilizatorul. În versiunile mai vechi de Windows și în Windows NT 3.51, pictogramele există în principal în *Program Manager*.

O aplicație obișnuită în Windows 95 sau Windows NT va avea cel puțin două pictograme: o pictogramă mare (32x32) și o pictogramă mică (16x16). Când aplicația este minimizată, Windows va afișa o pictogramă mică în colțul din stânga sus al ferestrei aplicației. Windows utilizează pictogramele mari pentru *desktop* și în vizualizările tip *Large Icons*.

În mod obișnuit, pictogramele se creează în editorul de imagini din Windows SDK sau cu un alt editor cum ar fi editorul de pictograme din *Visual C++*. Apoi se utilizează instrucțiunea *ICON* pentru adăugarea pictogramelor produse într-un fișier resursă de tip script al aplicației. Un exemplu tipic de folosire a pictogramei se află în rutina de înregistrare a clasei fereastră principală. Așa cum ați observat de-a lungul celor 187 de secțiuni precedente, programele vor înregistra o pictogramă la înregistrarea clasei fereastră prin apelul la *RegisterClassEx*, ca mai jos:

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE
                    hPrevInstance, LPTSTR lpCmdLine,
                    int nCmdShow)
  {
    MSG msg;
    HWND hWnd;
    WNDCLASS wclass;

    // Inregistreaza clasa fereastra principala a aplicatiei.
    //.........................................
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
```

```
wc.hIcon = LoadIcon( hInstance, lpszAppName );
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName = lpszAppName;
wc.lpszClassName = lpszAppName;

if ( !RegisterClass( &wc ) )
  return( FALSE );
// alte instructiuni ...
```

Așa cum observați, fragmentul de cod de mai sus înregistrează o pictogramă cu numele conținut în șirul *lpszAppName*, ca pictogramă a ferestrei principale. În următoarele secțiuni, veți învăța mai mult despre modul în care puteți atașa o pictogramă la o aplicație.

1445 CREAREA PICTOGRAMELOR

C/C++

Ca și în cazul blocurilor bitmap și al metafișierelor, Windows permite producerea și modificarea pictogramelor în timpul rulării. Programele dumneavoastră vor utiliza funcția *CreateIcon* pentru producerea pictogramelor din matrice binare, din datele aparținând unui bitmap sau din blocurile bitmap independente de dispozitiv. Funcția *CreateIcon* creează o pictogramă cu dimensiuni, culori și modele de biți specificate. Veți implementa funcția *CreateIcon* potrivit prototipului prezentat mai jos:

```
HICON CreateIcon(
    HINSTANCE hInstance,      // identificator pentru instanta
                              // aplicatiei
    int nWidth,               // latime pictograma
    int nHeight,              // inaltime pictograma
    BYTE cPlanes,             // numar planuri din masca de
                              // biti XOR
    BYTE cBitsPixel,          // numar de biti pe pixel din
                              // masca de biti XOR
    CONST BYTE *lpbANDbits,   // pointer la matricea cu masca de
                              // biti AND
    CONST BYTE *lpbXORbits    // pointer la matricea cu masca
                              // de biti XOR
);
```

Funcția *CreateIcon* acceptă parametrii prezentați în Tabelul 1445.1.

Parametru	Descriere
hInstance	Identifică instanța modulului care produce pictograma.
nWidth	Specifică lățimea pictogramei, în pixeli.
nHeight	Specifică înălțimea pictogramei, în pixeli.
cPlanes	Specifică numărul de planuri din masca de biți XOR a pictogramei.
cBitsPixel	Specifică numărul de biți pe pixel din masca de biți XOR a pictogramei.

Parametru	Descriere
lpbANDbits	Indică o matrice de octeți care conține valorile biților din masca de biți AND a pictogramei. Această mască descrie un bitmap monocrom.
lpbXORbits	Indică o matrice de octeți care conține valorile biților din masca de biți XOR a pictogramei. Această mască descrie un bitmap monocrom sau un bitmap color dependent de dispozitiv.

***Tabelul 1445.1** Parametrii funcției **CreateIcon**.*

Parametrii *nWidth* și *nHeight* trebuie să specifice lățimea și înălțimea acceptate de driverul dispozitivului curent de afișare, deoarece sistemul nu poate produce pictograme de alte dimensiuni. Pentru obținerea lățimii și înălțimii acceptate de driverul dispozitivului de afișare, utilizați funcția *GetSystemMetrics*, specificând valorile *SM_CXICON* sau *SM_CYICON*.

Funcția *CreateIcon* produce pictograma din două blocuri bitmap (pe care funcția le folosește ca măști de biți), masca de biți AND și cea XOR. Masca de biți AND este întotdeauna un bitmap monocrom, cu un bit pe pixel. Funcția *CreateIcon* aplică următoarea tabelă de adevăr măștilor de biți *AND* și *XOR*, prezentată în Tabelul 1445.2.

Masca de biți AND	Masca de biți XOR	Afișare
0	0	Negru
0	1	Alb
1	0	Ecran
1	1	Ecran inversat

***Tabelul 1445.2** Valorile de adevăr pe care funcția **CreateIcon** le aplică măștilor de biți **AND** și **XOR**.*

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateIcon*, analizați programul *Create_Icon.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Create_Icon.cpp* specifică în mod direct valorile biților din măștile de biți *AND* și *XOR* ale pictogramei pentru a produce o pictogramă monocromă. Când rutina *WndProc* primește mesajul *WM_CREATE* programul produce o pictogramă; când aceeași rutină primește mesajul *WM_PAINT*, programul trasează pictograma pe ecran.

CREAREA UNEI PICTOGRAME DINTR-O RESURSĂ C/C++ 1446

Programele dumneavoastră pot produce pictograme în mai multe feluri. Însă, de regulă, programele nu vor produce două blocuri bitmap și două măști de biți în memorie, ca în programul *Create_Icon.cpp* prezentat în secțiunea 1445. La fel ca și în cazul tabelelor de șiruri și alte informații reutilizabile, programul dumneavoastră poate încărca biții componenți ai unei pictograme dintr-un fișier de resurse și converti biții într-o pictogramă reală. Pentru efectuarea acestei prelucrări, programul va utiliza funcția *CreateIconFromResource*. Această funcție produce o pictogramă sau un cursor din biții resursă care descriu pictograma. Veți implementa funcția *CreateIconFromResource* potrivit prototipului prezentat mai

```
HICON CreateIconFromResource(
     PBYTE presbits,   // pointer la bitii pictogramei sau
                       // cursorului
     DWORD dwResSize,  // numarul de octeti din bufferul de biti
     BOOL fIcon,       // indicator de pictograma sau cursor
     DWORD dwVer       // verisune de format Windows
);
```

Funcția *CreateIconFromResource* acceptă parametrii prezentați în Tabelul 1446.1.

Parametru	Descriere
presbits	Indică bufferul care conține biții resursei pictogramei sau cursorului. Apelurile la funcțiile *LookupIconIDFromDirectory* (în Windows 95, puteți apela și *LookupIconIDFromDirectoryEx*) și *LoadResource* încarcă de obicei acești biți.
DwResSize	Specifică dimensiunea, în octeți, a setului de biți indicat de parametrul *presbits*.
fIcon	Specifică dacă se va crea o pictogramă sau un cursor. Dacă parametrul este TRUE, se va crea o pictogramă, iar dacă este FALSE, se va crea un cursor.
dwVer	Specifică numărul versiunii formatului de pictogramă sau cursor pentru resursele de biți la care indică parametrul *presbits*.

Tabelul 1446.1 *Parametrii funcției* ***CreateIconFromResource***.

Parametrul *dwVer* poate lua una din valorile enumerate în Tabelul 1446.2.

Format	dwVer
Windows 2.x	0x00020000
Windows 3.x	0x00030000

Tabelul 1446.2 *Valorile parametrului* ***dwVer***.

Toate aplicațiile bazate pe Microsoft Win32 utilizează formatul Windows 3.x pentru pictograme și cursoare. Funcțiile *CreateIconFromResource, CreateIconIndirect, GetIconInfo* și *LookupIconIdFromDirectory* (și, în Windows 95, funcțiile *CreateIconFromResourceEx* și *LookupIconIdFromDirectoryEx*) permit aplicațiilor de tip shell și browserelor să examineze și să utilizeze resursele din întreg sistemul.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateIconFromResource*, analizați programul *Display_Res_Icon.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *Display_Res_Icon.cpp* utilizează diverse funcții de gestionare a resurselor pentru localizarea unei pictograme într-un fișier de resurse, pentru găsirea locației fișierului pe disc și încărcarea pictogramei în memorie. Programul apoi afișează pictograma în zona client a ferestrei. Prelucrarea operativă are loc după ce utilizatorul alege opțiunea *Test!*, iar codul care efectuează prelucrarea se află în funcția *WndProc*.

1447 UTILIZAREA FUNCȚIEI CREATEICONINDIRECT C/C++

Programele dumneavoastră pot crea pictograme din blocuri bitmap sau dintr-un identificator de resurse. De asemenea, programele pot produce pictograme dintr-o valoare de structură. În acest scop, veți utiliza funcția *CreateIconIndirect* pentru crearea pictogramelor din

componentele nedefinite în program sau în fișierele de resurse. Funcția *CreateIconIndirect* produce o pictogramă sau un cursor dintr-o structură *ICONINFO*. Veți implementa funcția *CreateIconIndirect* potrivit prototipului prezentat mai jos:

```
HICON CreateIconIndirect(PICONINFO piconinfo);
```

Parametrul *piconinfo* indică o structură *ICONINFO* utilizată de funcție pentru a produce o pictogramă sau un cursor. Dacă funcția reușește, valoarea de returnare este identificatorul pictogramă sau cursorul produs. Sistemul copiază blocurile bitmap în structura *ICONINFO* înaintea producerii pictogramei sau cursorului. Aplicațiile trebuie să continue să gestioneze blocurile bitmap inițiale și să le șteargă atunci când nu mai sunt necesare. Structura *ICONINFO* conține informații despre o pictogramă sau un cursor. Windows API definește structura *ICONINFO* în modul prezentat mai jos:

```
typedef struct _ICONINFO {
    BOOL fIcon;
    DWORD xHotspot;
    DWORD yHotspot;
    HBITMAP hbmMask;
    HBITMAP hbmColor;
} ICONINFO;
```

Tabelul 1447 prezintă membrii structurii ***ICONINFO***.

Membru	Descriere
fIcon	Arată dacă structura definește o pictogramă sau un cursor. Valoarea TRUE specifică o pictogramă; valoare FALSE, specifică un cursor.
xHotspot	Conține coordonata x a punctului activ al unui cursor. Dacă structura definește o pictogramă, punctul activ este întotdeauna în centrul pictogramei, iar acest membru este ignorat.
yHotspot	Conține coordonata y a punctului activ al unui cursor. Dacă structura definește o pictogramă, punctul activ este întotdeauna în centrul pictogramei, iar acest membru este ignorat.
hbmMask	Specifică blocul bitmap cu masca de biți a pictogramei. Dacă structura definește o pictogramă alb-negru, această mască de biți este formatată astfel încât jumătatea de sus să fie masca de biți AND, iar jumătatea de jos, masca de biți XOR ai pictogramei. În această situație, înălțimea va fi un multiplu de doi. Dacă structura definește o pictogramă color, masca definește numai biții AND ai pictogramei.
hbmColor	Identifică blocul bitmap color al pictogramei. Acest membru poate fi opțional dacă structura definește o pictogramă alb-negru. Masca de biți AND din membrul *hbmMask* se aplică indicatorului *SRCAND* la destinație. Apoi, *CreateIconIndirect* folosește indicatorul *SRCINVERT* pentru a aplica blocul bitmap color (folosind XOR) la destinație.

Tabelul 1447 *Membrii structurii* ***ICONINFO***.

Pe scurt, structura *ICONINFO* definește blocurile bitmap monocrom și color, iar funcția *CreateIconIndirect* combină blocurile bitmap pe baza valorilor din structura *ICONINFO*. Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateIconIndirect*, analizați programul *Two_Icons.cpp*, conținut pe CD-ROM-ul care însoțește cartea de față. Programul

Two_Icons.cpp combină două blocuri bitmap într-un al treilea bitmap pictogramă. Ca de obicei, programul handler pentru mesajul *WM_COMMAND* din funcția *WndProc* se ocupă de prelucrarea principală.

1448 UTILIZAREA FUNCȚIEI LOADICON



În secțiunile precedente ați învățat modurile în care programele dumneavoastră pot utiliza diferite metode pentru producerea pictogramelor în timpul rulării. Însă, așa cum ați observat în alte programe, programele dumneavoastră vor utiliza în mod obișnuit funcția *LoadIcon* pentru încărcarea în program a unei pictograme din fișierul de resurse al programului. Funcția *LoadIcon* oferă programului o modalitate simplă și eficientă de încărcare a pictogramelor anterior create în alte programe. Funcția *LoadIcon* încarcă resursa pictogramă specificată din fișierul executabil (.EXE) asociat cu instanța aplicației. Veți implementa funcția *LoadIcon* potrivit prototipului prezentat mai jos:

```
HICON LoadIcon(
    HINSTANCE hInstance, // identificator pentru instanta
                         // aplicatiei
    LPCTSTR lpIconName   // sirul cu numele pictogramei sau
                         // identificatorul de resursa al
                         // pictogramei
);
```

Parametrul *hInstance* identifică instanța modulului al cărui fișier executabil conține pictograma care urmează să fie încărcată. Acest parametru trebuie să fie NULL când este încărcată o pictogramă standard. Parametrul *lpIconName* indică un șir terminat cu NULL care conține numele resursei pictogramă care urmează să fie încărcată. Sau, acest parametru poate conține identificatorul de resursă în cuvântul mai puțin semnificativ și zero în cuvântul mai semnificativ. Pentru crearea acestei valori de identificator de resursă utilizați macrocomanda *MAKEINTRESOURCE*. Pentru a utiliza una din pictogramele predefinite în Windows, fixați parametrul *hInstance* la NULL, iar parametrul *lpIconName* la una din valorile descrise înTabelul 1448.

Valoare	Descriere
IDI_APPLICATION	Pictograma implicită a aplicației.
IDI_ASTERISK	Asterisc (utilizat în mesajele conținând informații).
IDI_EXCLAMATION	Semnul de exclamare (utilizat în mesajele conținând avertismente).
IDI_HAND	Pictograma cu reprezentarea unei mâini (utilizat în mesajele conținând avertismente grave).
IDI_QUESTION	Semnul de întrebare (utilizat în mesajele conținând cereri de opțiuni).
IDI_WINLOGO	Emblema Windows.

Tabelul 1448 *Pictogramele predefinite în Windows.*

Funcția *LoadIcon* încarcă resursa pictogramă numai dacă nu a fost încărcată anterior; altfel, ea preia identificatorul pentru resursa existentă. Funcția caută resursa pictogramă pentru cea mai potrivită pictogramă pentru afișarea curentă. Resursa pictogramă poate fi un bitmap color sau monocrom. *LoadIcon* poate numai să încarce o pictogramă ale cărei dimensiuni

corespund cu valorile *SM_CXICON* și *SM_CYICON* ale sistemului metric. Pentru încărcarea unor pictograme de alte dimensiuni utilizați *LoadImage*.

UTILIZAREA FUNCȚIEI LOADIMAGE PENTRU ÎNCĂRCAREA MAI MULTOR TIPURI GRAFICE

C/C++ 1449

Programele dumneavoastră pot utiliza *LoadIcon* pentru încărcarea unei pictograme din fișierul de resurse al programului. De asemenea, pot fi utilizate funcțiile *LoadBitmap* și *LoadCursor* pentru încărcarea imaginilor de tip bitmap și respectiv cursor din fișierul de resurse al programului. Sau, se poate utiliza funcția *LoadImage* care încarcă o pictogramă, un cursor sau un bitmap. Veți implementa funcția *LoadImage* potrivit prototipului prezentat mai jos:

```
HANDLE LoadImage(
    HINSTANCE hinst,    // identificator pentru instanta care
                        // contine imaginea
    LPCTSTR lpszName,   // nume sau identificator al imaginii
    UINT uType,         // tip imagine
    int cxDesired,      // latime dorita
    int cyDesired,      // inaltime dorita
    UINT fuLoad         // indicatoare de incarcare
);
```

Funcția *LoadImage* acceptă parametrii prezentați în Tabelul 1449.1.

Parametru	Descriere
hinst	Identifică o instanță a modulului care conține imaginea ce trebuie încărcată. Pentru încărcarea unei imagini OEM, fixați acest parametru la zero.
lpszName	Identifică imaginea care trebuie încărcată. Dacă parametrul *hinst* nu este NULL și parametrul *fuLoad* nu conține *LR_LOADFROMFILE*, atunci *lpszName* este un pointer la un șir terminat cu NULL, care cuprinde numele resursei imagine din modulul *hinst*. Dacă *hinst* este NULL și nu este specificat *LR_LOADFROMFILE*, atunci cuvântul mai puțin semnificativ al acestui parametru trebuie să fie identificatorul pentru imaginea OEM de încărcat. Identificatorii imaginii OEM sunt definiți în fișierul WINUSER.H și au prefixele enumerate în Tabelul 1449.2. Sub Windows 95, dacă parametrul *fuLoad* include valoarea *LR_LOADFROMFILE* atunci *lpszName* este numele fișierului care conține imaginea. Sub Windows NT, *LR_LOADFROMFILE* nu este acceptat.
uType	Specifică tipul imaginii ce urmează să se încarce. Acest parametru poate lua una din valorile enumerate în Tabelul 1449.3.
cxDesired	Specifică lățimea pictogramei sau cursorului, în pixeli. Dacă parametrul este zero, iar parametrul *fuLoad* este *LR_DEFAULTSIZE*, funcția utilizează valorile *SM_CXICON* sau *SM_CXCURSOR* din sistemul metric pentru stabilirea lățimii. Dacă acest parametru este zero și *LR_DEFAULTSIZE* nu este utilizat, funcția folosește lățimea reală din resursă.

(continuare)

Parametru	Descriere
cyDesired	Specifică înălțimea pictogramei sau cursorului, în pixeli. Dacă parametrul este zero, iar parametrul *fuLoad* este *LR_DEFAULTSIZE*, funcția utilizează valorile *SM_CYICON* sau *SM_CYCURSOR* din sistemul metric pentru stabilirea înălțimii. Dacă acest parametru este zero și *LR_DEFAULTSIZE* nu este utilizat, funcția folosește înălțimea reală din resursă.
fuLoad	Specifică modul în care funcția încarcă imaginea. Parametrul este o combinație a valorilor descrise în Tabelul 1449.4.

Tabelul 1449.1 *Parametrii funcției* ***LoadImage***.

Așa cum indică Tabelul 1449.1, programele pot încărca imagini OEM. În acest caz, trebuie specificat unul din identificatorii enumerați în Tabelul 1449.2.

Prefix	Semnificație
*OBM_*	Blocuri bitmap OEM
*OIC_*	Pictograme OEM
*OCR_*	Cursoare OEM

Tabelul 1449.2 *Identificatorii imaginii.*

Așa cum ați învățat, programele pot încărca unul sau mai multe tipuri de imagine cu funcția *LaodImage*. Parametrul *uType* specifică tipul de imagine, care poate fi una din valorile descrise în Tabelul 1449.3.

Valoare	Semnificație
IMAGE_BITMAP	Încarcă un bitmap.
IMAGE_CURSOR	Încarcă un cursor.
IMAGE_ICON	Încarcă o pictogramă.

Tabelul 1449.3 *Tipuri posibile de imagine.*

Puteți încărca fișierul de imagini cu mai multe opțiuni de afișare. Parametrul *fuLoad* trebuie să fie una sau o combinație a valorilor descrise în Tabelul 1449.4.

Valoare	Semnificație
LR_DEFAULTCOLOR	Indicator implicit. Înțelesul lui este „nu *LR_MONOCHROME*".
LR_CREATEDIBSECTION	Când parametrul *uType* specifică *IMAGE_BITMAP*, determină funcția să returneze o secțiune de bitmap DIB și nu un bitmap compatibil. Acest indicator este util pentru încărcarea unui bitmap fără maparea lui la culorile dispozitivului de afișare.
LR_DEFAULTSIZE	Utilizează lățimea și înălțimea specificate de valorile sistemului metric pentru cursoare și pictograme, dacă *cxDesired* și *cyDesired* sunt fixate pe zero. Dacă acest indicator nu este specificat, iar *cxDesired* și *cyDesired* nu sunt fixate pe zero, funcția utilizează dimensiunea reală a resursei. Dacă resursa conține mai multe imagini, funcția utilizează dimensiunea primei imagini.
LR_LOADFROMFILE	Încarcă imaginea din fișierul specificat de parametrul *lpszName*. Dacă indicatorul nu este specificat, *lpszName* va fi numele resursei.

Valoare	Semnificație
LR_LOADMAP3DCOLORS	Caută tabela de culoare pentru imagine și înlocuiește umbrele de gri cu culorile tridimensionale corespunzătoare, arătate în Tabelul 1449.5.
LR_LOADTRANSPARENT	Regăsește valoarea de culoare a primului pixel din imagine și înlocuiește intrarea respectivă din tabela de culoare cu culoarea implicită a ferestrei (*COLOR_WINDOW*). Toți pixelii din imagine care folosesc acea intrare preiau culoarea implicită. Această valoare se aplică numai imaginilor care au tabele de culoare corespunzătoare. Dacă *fuLoad* include atât valoarea *LR_LOADTRANSPARENT*, cât și *LR_LOADMAP3DCOLORS*, atunci *LRLOADTRANSPARENT* este prevalentă. Însă tabela de culoare este înlocuită de funcția *afuLoad* cu *COLOR_3DFACE* și nu cu *COLOR_WINDOW*.
LR_MONOCHROME	Încarcă imaginea în alb-negru.
LR_SHARED	Partajează identificatorul imaginii dacă imaginea este încărcată de mai multe ori simultan, de către unul sau mai multe programe. Dacă *LR_SHARED* nu este activat, un al doilea apel la *LoadImage* pentru aceeași resursă, va încărca imaginea din nou și va returna un alt identificator. Nu utilizați *LR_SHARED* pentru imagini care au dimensiuni ne-standard și se pot modifica după încărcare sau care sunt încărcate dintr-un fișier.

Tabelul 1449.4 *Opțiunile de încărcare pentru fișierul imagine.*

Când încărcați un fișier imagine trifimensională, Windows API va mapa (converti) culorile în locul dumneavoastră. Windows va înlocui fiecare culoare din coloana stângă a Tabelului 1449.5 cu culoarea respectivă din coloana dreaptă.

Coloare	Înlocuită cu
Gri închis, RGB(128,128,128)	*COLOR_3DSHADOW*
Gri, RGB(192,192,192)	*COLOR_3DFACE*
Gri deschis, RGB(223,223,223)	*COLOR_3DLIGHT*

Tabelul 1449.5 *Valorile de mapare 3D pentru* ***LoadImage***.

Așa cum observați, programele dumneavoastră pot utiliza *LoadImage* în orice situație în care ar putea utiliza *LoadIcon, LoadBitmap sau LoadCursor.*

OPERAȚIILE DE INTRARE/IEȘIRE (I/O) CU FIȘIERE ÎN WINDOWS

C/C++1450

În capitolele anterioare ale acestei cărți ați învățat despre operațiile de intrare/ieșire (I/O) cu fișiere. Ați aflat cum să efectuați operațiile de I/O cu fișiere atât în C, cât și în C++. În următorul capitol veți învăța fundamentele operațiilor de I/O cu fișiere în Windows.

Așa cum știți, conceptul tradițional de fișier constă într-un bloc de date pe un dispozitiv de stocare. Blocul de date este identificat de un identificator unic care este numele fișierului. În mediul DOS, programul va salva de obicei fișierele pe disc sau pe alte unități. În schimb, pentru operațiile I/O, interfața Win32 API tratează „fișierul" ca fișier pe disc, canal de transfer nominal (*named pipe*), resurse de comunicații, unități de disc sau consolă de intrare sau

ieșire. Fiecare tip de fișier este identic la nivelul de bază, dar fiecare are propriile caracteristici și limitări. Funcțiile Win32 API pentru operații cu fișiere permit programelor să acceseze fișierele indiferent de sistemul de fișiere care stă la bază sau tipul dispozitivelor.

1451 CANALELE DE TRANSFER, RESURSELE, DISPOZITIVELE ȘI FIȘIERELE



Așa cum ați învățat în secțiunea 1450, Windows asigură suport pentru operații de intrare/ieșire de tip fișier pe o varietate largă de dispozitive. În secțiunile următoare nu veți învăța numai despre fișierele standard de intrare și ieșire de pe disc, ci și despre operații fundamentale de intrare/ieșire pe alte dispozitive. E util să cunoașteți unele din cele mai obișnuite dispozitive pe care le veți întâlni când lucrați cu programe sub Windows. Tabelul 1451 enumeră o parte din cele mai obișnuite dispozitive.

Dispozitiv	Utilizare obișnuită
Fișier	Stocare persistentă de date
Director	Ansambluri de atribute și fișiere
Unitate Logică de Disc	Formatare
Unitate Fizică de Disc	Acces la tabela de partiție
Port serial	Transmitere de date prin linie telefonică
Port paralel	Transmitere de date către imprimantă
Slot pentru mail	Transmitere de date unu-la-mai-mulți, de obicei în rețea, către o mașină Windows
Canal de transfer nominal	Transmitere de date unu-la-unu, de obicei în rețea, către o mașină Windows.
Canal de transfer anonim (*anonymous pipe*)	Transmitere de date unu-la-unu pe o singură mașină (niciodată în rețea)
Soclu (Socket)	Transmisie de date sub formă de flux sau datagrame, de obicei în rețea, către orice mașină care acceptă soclu (mașina nu este exclusiv Windows)
Consolă	Un buffer ecran cu fereastră de text

***Tabelul 1451** Dispozitivele obișnuite și utilizarea lor.*

După cum veți afla, Win32 încearcă să ascundă diferențele dintre dispozitive, pentru utilizator, pe cât posibil. Cu alte cuvinte, dacă deschideți un slot pentru mail și un fișier, Windows vă va permite, în general, să citiți și să scrieți în oricare din dispozitive cu funcții similare.

E bine să remarcați însă, că în afară de *CreateFile, ReadFile* și *WriteFile,* Windows oferă o colecție extinsă de funcții de dispozitiv specifice care permit programelor dumneavoastră să gestioneze îndeaproape proprietățile specifice ale dispozitivelor. De exemplu, nu are nici un sens să stabiliți o rată de transmisie (*baud rate*) când utilizați un canal de transfer nominal pentru comunicare, dar are sens să procedați astfel când utilizați un port de comunicare. În consecință, funcția *SetCommConfig* va opera cu un dispozitiv de port serial, dar nu va opera corect cu un canal de transfer nominal. Din acest motiv, majoritatea secțiunilor care urmează se vor concentra asupra utilizării generale a funcției *CreateFile, ReadFile* și *WriteFile* și nu asupra aplicațiilor specifice ale funcțiilor pe un dispozitiv dat. Consultați documentația

compilatorului şi a dispozitivului de care dispuneţi, pentru informaţii suplimentare privind comunicarea cu un anumit dispozitiv.

UTILIZAREA FUNCŢIEI CREATEFILE PENTRU DESCHIDEREA FIŞIERELOR

C/C++1452

Interfaţa Win32 API asigură suport pentru multe tipuri de dispozitive şi vă permite manipularea fişierelor în programele dumneavoastră, pe orice tip de dispozitiv. Pentru a crea un fişier pe orice dispozitiv, programul va utiliza funcţia Win32 API *CreateFile*. Funcţia *CreateFile* creează sau deschide obiectele enumerate mai jos şi returnează un identificator ce poate fi utilizat pentru accesul la acel obiect:

- fişiere
- canale de transfer
- sloturi pentru mail
- resurse de comunicaţie
- dispozitive de disc (numai în Windows NT)
- console
- directoare (numai pentru deschidere)

Veţi implementa funcţia *CreateFile* potrivit prototipului prezentat mai jos:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,       // pointer la numele fisierului
    DWORD dwDesiredAccess,  // mod acces (citire-scriere)
    DWORD dwShareMode,        // mod partajare
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // atribute
                                               // securitate
    DWORD dwCreationDistribution, // mod creare
    DWORD dwFlagsAndAttributes,   // atribute fisiere
    HANDLE hTemplateFile          // identificator pentru
                                  // fisierul cu sablon
);
```

Funcţia *CreateFile* conferă programului dumneavoastră un control semnificativ asupra fişierului creat. Tabelul 1452.1 prezintă parametrii acceptaţi de funcţia *CreateFile*.

Parametru	Descriere
lpFileName	Indică un şir terminat cu NULL care specifică numele unui obiect pentru crearea sau deschidere (fişier, canal de transfer, slot pentru mail, unitate de disc, consolă sau director). Dacă **lpFileName* este o cale de acces, există o limită implicită pentru dimensiunea şirului de *MAX_PATH* caractere. Această limită este dependentă de modul în care funcţia *CreateFile* analizează căile de acces.

(continuare)

Parametru	Descriere
DwDesiredAccess	Specifică tipurile de acces la obiect. Aplicațiile pot obține acces la citire, acces la scriere, acces la scriere-citire sau accesul de interogare de unitate. Acest parametru poate fi orice combinație a valorilor prezentate în Tabelul 1452.2
DwShareMode	Stabilește indicatoarele pe biți care specifică modul de partajare al obiectului. Dacă *dwShareMode* este 0, obiectul nu poate fi partajat. Operațiile de deschidere ulterioare ale obiectului vor eșua până când programul care utilizează obiectul va închide identificatorul. Pentru partajarea obiectului, utilizați una din valorile sau o combinație a valorilor prezentate în Tabelul 1452.3
LpSecurityAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care decide dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpSecurityAttributes* este NULL, identificatorul nu poate fi moștenit. Windows NT: membrul *lpSecurityDescriptor* al structurii precizează descriptorul de securitate pentru obiect. Dacă *lpSecurityAttributes* este NULL, obiectul obține un descriptor de securitate implicit. Pentru ca acest parametru să aibă efect asupra fișierelor, sistemul de fișiere țintă trebuie să asigure suport pentru securitatea fișierelor și directoarelor. Windows 95: *CreateFile* ignoră membrul *lpSecurityDescriptor*.
DwCreationDistribution	Specifică ce acțiuni trebuie luate pentru fișierele existente sau pentru cele inexistente. Parametrul trebuie să ia una din valorile descrise în Tabelul 1452.4.
DwFlagsAndAttributes	Specifică atributele fișierului și indicatoarele pentru fișier. Orice combinație a atributelor enumerate în Tabelul 1452.5 este acceptabilă pentru parametrul *dwFlagsAndAttributes* cu excepția faptului că toate celelalte atribute de fișier vor suprascrie atributul *FILE_ATTRIBUTE_NORMAL*. Dacă funcția *CreateFile* deschide extremitatea client a unui canal de transfer nominal, parametrul *dwFlagsAndAttributes* poate conține și informațiile Security Quality of Service (SQOS). Secțiunea 1453 explică în detaliu deschiderea canalelor de transfer nominale. Când aplicația apelantă specifică indicatorul *SECURITY_SQOS_PRESENT*, parametrul *dwFlagsAndAttributes* conține una sau mai multe din valorile enumerate în Tabelul 1453.1.
HTemplateFile	Specifică un identificator cu acces *GENERIC_READ* la un fișier șablon. Acest fișier șablon furnizează atributele de fișier și atributele extinse pentru fișierul pe cale de a fi creat. Sub Windows 95, această valoare trebuie să fie NULL. Dacă furnizați un identificator sub Windows 95, apelul va eșua.

Tabelul 1452.1 *Parametrii funcției* ***CreateFile***.

Așa cum ați aflat din Tabelul 1452.1 programele pot specifica nivelul de acces dorit la fișierele deschise cu *CreateFile* în cadrul parametrului *dwAccess*. Valorile posibile ale lui *dwAccess* sunt prezentate în Tabelul 1452.2

Valoare	Semnificaţie
0	Specifică accesul cu interogare de unitate la obiect. O aplicaţie poat interoga atributele de unitate fără a avea acces la unitate.
GENERIC_READ	Specifică accesul cu citire la obiect. Datele pot fi citite din fişier, iar pointerul de fişier poate fi deplasat de apeluri API. Se combină cu *GENERIC_WRITE* pentru acces de citire-scriere.
GENERIC_WRITE	Specifică accesul cu scriere la obiect. Datele pot fi scrise în fişier, iar pointerul de fişier poate fi deplasat de apelurile API. Se combină cu *GENERIC_READ* pentru acces de citire-scriere.

Tabelul 1452.2 *Valorile posibile ale parametrului* ***dwAccess***.

Pe lângă nivelurile de acces, programele pot specifica un mod partajat pentru fişier în cadr parametrului *dwShareMode*. Tabelul 1452.3 enumeră valorile posibile pentru parametr *dwShareMode*.

Valoare	Semnificaţie
FILE_SHARE_DELETE	Numai Windows NT: operaţiile ulterioare de deschidere ale obiectului vor reuşi numai dacă este cerut accesul la ştergere.
FILE_SHARE_READ	Operaţiile ulterioare de deschidere ale obiectului vor reuşi numai dacă este cerut accesul la citire.
FILE_SHARE_WRITE	Operaţiile ulterioare de deschidere ale obiectului vor reuşi numai dacă este cerut accesul la scriere.

Tabelul 1452.3 *Valorile posibile ale parametrului* ***dwShareMode***.

Parametrul *dwCreate* stabileşte acţiunea pe care o va executa Windows când fişierı respectiv există sau nu. Parametrul va lua una din valorile enumerate în Tabelul 1452.4.

Valoare	Semnificaţie
CREATE_NEW	Creează un fişier nou. Funcţia eşuează dacă fişierul specificat există deja.
CREATE_ALWAYS	Creează un fişier nou. Funcţia suprascrie fişierul dacă el există.
OPEN_EXISTING	Deschide un fişier. Funcţia eşuează dacă fişierul nu există.
OPEN_ALWAYS	Deschide un fişier dacă el există. Dacă fişierul nu există, funcţia creează un fişier ca şi când *dwCreationDistribution* ar fi *CREATE_NEW*.
TRUNCATE_EXISTING	Deschide un fişier. O dată deschis, fişierul este trunchiat astfel încât dimensiunea sa să fie de zero octeţi. Procesul apelant trebuie să deschidă fişierul cu accesul cel puţin *GENERIC_WRITE*. Funcţia fişier eşuează dacă fişierul nu există.

Tabelul 1452.4 *Valorile posibile ale parametrului* ***dwCreate***.

Când creaţi un fişier în Windows, sistemul de operare Windows va ataşa atribute fişierulı respectiv. Puteţi preciza indicatoare şi atribute pentru fiecare fişier nou creat. Tabelul 1452. enumeră atributele şi indicatoarele posibile.

Atribut	Semnificație
FILE_ATTRIBUTE_ARCHIVE	Marchează fișierul pentru arhivare. Aplicațiile folosesc acest atribut pentru marcarea fișierelor pentru salvare de siguranță sau eliminare.
FILE_ATTRIBUTE_COMPRESSED	Fișierul sau directorul vor fi comprimate. În cazul fișierului aceasta înseamnă că toate datele din fișier sunt comprimate. În cazul directorului, comprimarea este implicită pentru fișierele sau directoarele nou create.
FILE_ATTRIBUTE_HIDDEN	Fișierul este ascuns. El nu va fi cuprins în enumerarea obișnuită a conținutului unui director.
FILE_ATTRIBUTE_NORMAL	Fișierul nu are stabilite alte atribute. Acest atribut este valid numai dacă este folosit singur la apelarea funcției *CreateFile*.
FILE_ATTRIBUTE_OFFLINE	Datele din fișier nu sunt imediat disponibile. Precizează că datele din fișier au fost mutate fizic într-o stocare *offline*.
FILE_ATTRIBUTE_READONLY	Fișierul este protejat la scriere (*read-only*). Aplicațiile pot citi fișierul dar nu pot efectua operații de scriere sau ștergere.
FILE_ATTRIBUTE_SYSTEM	Fișierul este o parte a sistemului de operare sau este utilizat exclusiv de către acesta.
FILE_ATTRIBUTE_TEMPORARY	Un proces utilizează fișierul pentru stocare temporară. Un fișier temporar trebuie șters de aplicație imediat ce nu mai este nevoie de el.
FILE_FLAG_WRITE_THROUGH	Comunică sistemului să scrie prin orice memorie intermediară *cache* direct pe disc. Windows poate menține operațiile de scriere în memoria *cache*, dar nu le poate transmite oricând.
FILE_FLAG_OVERLAPPED	Comunică sistemului de operare să inițializeze obiectul, astfel încât operațiile care necesită un volum mare de timp de prelucrare returnează *ERROR_IO_PENDING*. Când operația s-a terminat, evenimentul respectiv este pus pe starea de semnalizare în structura *OVERLAPPED*. Când specificați *FILE_FLAG_OVERLAPPED*, funcțiile *ReadFile* și *WriteFile* trebuie să specifice o structură *OVERLAPPED*. Aceasta înseamnă că atunci când este specificat *FILE_FLAG_OVERLAPPED*, aplicația trebuie să efectueze operații de scriere și citire suprapuse. Când specificați *FILE_FLAG_OVERLAPPED*, sistemul nu mai menține pointerul de fișier. Poziția fișierului trebuie transmisă de procesul apelant ca parte a parametrului *lpOverlapped* (indicând structura *OVERLAPPED*) către funcțiile *ReadFile* și *WriteFile*. Acest indicator permite procesului să execute mai multe operații simultan, cu un singur identificator (de exemplu, operația simultană de citire și scriere).

Atribut	Semnificaţie
FILE_FLAG_NO_BUFFERING	Comunică sistemului să deschidă fişierul fără buffere şi memorie *cache* intermediare. Când este combinat cu *FILE_FLAG_OVERLAPPED,* acest indicator conferă performanţă asincronă maximă deoarece I/O nu depind de operaţiile sincrone ale gestionării memoriei. Însă, unele operaţii I/O vor dura mai mult, pentru că datele nu sunt păstrate în memoria *cache.* Aplicaţiile trebuie să îndeplinească anumite cerinţe când lucrează cu fişiere deschise cu *FILE_FLAG_NO_BUFFERING.* Accesul la fişier trebuie să înceapă la deplasamentele în octeţi din cadrul fişierului, care sunt multipli întregi ai mărimii sectorului de volum. Accesul la fişier trebuie să fie pentru numărul de octeţi ce sunt multipli ai dimensiunii sectorului de volum. De exemplu, dacă mărimea sectorului este de 512 octeţi, o aplicaţie poate cere citirea şi scrierea a 512, 1024 sau 2048 de octeţi, dar nu a 335, 981 sau 7171 de octeţi. Bufferul adresat de operaţiile de citire şi scriere trebuie să fie aliniat la adresele de memorie care sunt multipli întregi ai dimensiunii sectorului de volum. Un mijloc de aliniere a bufferelor la multipli întregi ai dimensiunii sectorului de volum este utilizarea funcţiei *VirtualAlloc* pentru alocarea bufferelor. Ea alocă memorie aliniată la adresele care sunt multipli întregi ai dimensiunii paginii de memorie a sistemului de operare. Deoarece atât pagina de memorie, cât şi dimensiunile sectorului de volum sunt puteri ale lui 2, această memorie este, de asemenea, aliniată la adrese care sunt multipli întregi ai dimensiunii sectorului de volum. O aplicaţie poate stabili dimensiunea sectorului de volum apelând funcţia *GetDiskFreeSpace.*
FILE_FLAG_RANDOM_ACCESS	Precizează că procesul va avea acces aleator la fişier. Sistemul poate utiliza această valoare pentru optimizarea memoriei *cache* a fişierelor.
FILE_FLAG_SEQUENTIAL_SCAN	Precizează că procesul va accesa fişierul secvenţial, de la început la sfârşit. Sistemul poate utiliza această valoare ca mijloc de optimizare a memoriei *cache* a fişierelor. Dacă o aplicaţie deplasează pointerul de fişier pentru acces aleator, nu se va produce o memorare *cache* optimă; dar se garantează totuşi operarea corectă. Specificarea acestui indicator poate creşte performanţa aplicaţiilor care citesc fişiere mari folosind acces secvenţial. Performanţele realizate pot fi şi mai remarcabile pentru aplicaţiile care citesc fişiere mari preponderent secvenţial, dar ocazional sar peste mici intervale de octeţi.

(continuare)

Atribut	Semnificație
FILE_FLAG_DELETE_ON_CLOSE	Precizează că sistemul de operare va șterge fișierul imediat după închiderea tuturor identificatorilor săi, nu numai identificatorul pentru care ați specificat *FILE_FLAG_DELETE_ON_CLOSE*. Cererile de deschidere ulterioare pentru fișier vor eșua, dacă nu este utilizat indicatorul *FILE_SHARE_DELETE*.
FILE_FLAG_BACKUP_SEMANTICS	Numai pentru Windows NT: precizează că fișierul este deschis sau creat pentru operații de salvare de siguranță sau restaurare. Sistemul de operare asigură că procesul apelant suprascrie testele de securitate ale fișierului, dacă are permisiunea necesară să procedeze astfel. Permisiunile relevante sunt *SE_BACKUP_NAME* și *SE_RESTORE_NAME*. Puteți, de asemenea, să stabiliți acest indicator pentru obținerea unui identificator pentru director. Un identificator pentru director poate fi transmis unor funcții Win32 în locul unui identificator de fișier.
FILE_FLAG_POSIX_SEMANTICS	Precizează că fișierul va fi accesat potrivit regulilor *POSIX*. Aceasta permite denumiri similare pentru mai multe fișiere asigurând diferențierea numai prin literele mari și mici, pentru sisteme de fișiere care acceptă astfel de nume. Fiți prudenți la utilizarea acestei opțiuni, deoarece fișierele create cu această opțiune nu pot fi accesibile aplicațiilor scrise pentru MS-DOS sau Windows.

***Tabelul 1452.5** Indicatoarele și atributele posibile pentru parametrul **dwFlagsAndAttributes**.*

Dacă funcția *CreateFile* reușește, valoarea de returnare este un identificator deschis pentru fișierul specificat. Dacă fișierul respectiv există înainte de apelul funcției, iar *dwCreationDistribution* este *CREATE_ALWAYS* sau *OPEN_ALWAYS*, apelul la *GetLastError* returnează *ERROR_ALREADY_EXISTS* (chiar dacă funcția a reușit). Dacă fișierul nu există înaintea apelului, *GetLastError* returnează zero. Dacă funcția eșuează, valoarea de returnare este *INVALID_HANDLE_VALUE*.

Așa cum s-a remarcat mai sus, specificarea valorii zero pentru *dwDesiredAccess* permite aplicației să interogheze atributele de dispozitiv fără să aibă efectiv acces la unitate. Acest tip de interogare este util dacă, de exemplu, o aplicație urmărește să determine dimensiunea unei unități de disc flexibil și formatele pe care le acceptă, fără ca discul flexibil să existe în unitate.

Pentru a înțelege mai bine prelucrările efectuate de funcția *CreateFile*, analizați programul *First_File.cpp* conținut pe CD-ROM-ul care însoțește cartea de față. Programul *First_File.cpp* creează fișierul *file.dat* în care scrie un șir. Când utilizatorul selectează opțiunea *Test!*, programul va citi șirul din fișier și îl va afișa într-o casetă de mesaj.

1453 *UTILIZAREA FUNCȚIEI CREATEFILE CU DIFERITE DISPOZITIVE*



Programele dumneavoastră pot utiliza funcția *CreateFile* pentru crearea fișierelor pe o mare varietate de dispozitive. Există, însă, diferențe semnificative în modul în care *CreateFile*

operează, în funcţie de dispozitivul pe care programul îl transmite acestei funcţii. Următoarele paragrafe descriu unele din versiunile dependente de dispozitiv ale prelucrării funcţiei *CreateFile*.

Când creaţi un fişier nou, funcţia *CreateFile* efectuează următoarele acţiuni:

- Combină indicatoarele şi atributele de fişier specificate de parametrul *dwFlagsAndAttributes* cu valoarea *FILE_ATTRIBUTE_ARCHIVE*.
- Stabileşte lungimea fişierului la zero.
- Dacă parametrul *hTemplateFile* este specificat, copiază atributele extinse puse la dispoziţie de fişierul şablon pentru noul fişier,

Când funcţia *CreateFile* deschide un fişier existent, ea efectuează următoarele acţiuni:

- Combină indicatoarele fişierului specificat de *dwFlagsAndAttributes* cu atributele de fişier existente. *CreateFile* ignoră atributele de fişier specificate de *dwFlagsAndAttributes*.
- Stabileşte lungimea fişierului în funcţie de valoarea lui *dwCreationDistribution*.
- Ignoră parametrul *hTemplateFile*.
- Ignoră membrul *lpSecurityDescriptor* al structurii *SECURITY_ATTRIBUTES* dacă parametrul *lpSecurityAttributes* nu este NULL. *CreateFile* nu utilizează ceilalţi membri ai structurii. Membrul *bInheritHandle* reprezintă singurul mijloc de a preciza dacă identificatorul de fişier poate fi moştenit de alt proces.

Dacă încercaţi să creaţi un fişier pe o unitate de disc flexibil care nu conţine un disc sau pe o unitate de CD-ROM care nu are un CD, sistemul afişează o casetă de mesaj care cere utilizatorului să introducă un disc sau respectiv un CD. Pentru a preveni ca sistemul să afişeze această casetă de mesaj, apelaţi funcţia *SetErrorMode* cu *SEM_FAILCRITICALERRORS*. Pentru alte informaţii despre funcţia *SetErrorMode*, consultaţi documentaţia on-line a compilatorului.

Dacă funcţia *CreateFile* deschide extremitatea client a unui canal de transfer nominal, funcţia utilizează orice instanţă a canalului de transfer care se află în stare de recepţie. Procesul de deschidere poate duplica identificatorul de oricâte ori e nevoie, dar, o dată deschisă extremitatea client, instanţa canalului de transfer nominal nu poate fi deschisă de un alt client. Accesul specificat când canalul de transfer este deschis trebuie să fie compatibil cu accesul specificat în parametrul *dwOpenMode* al funcţiei *CreateNamedPipe*. Când precizaţi securitatea în conexiune cu deschiderea unui canal de transfer nominal, programul trebuie, de asemenea, să specifice unul sau mai multe indicatoare de securitate descrise în Tabelul 1453.1.

Valoare	Semnificaţie
SECURITY_ANONYMOUS	Precizează că fişierul va personifica clientul la nivelul Anonymous.
SECURITY_IDENTIFICATION	Precizează că fişierul va personifica clientul la nivelul Identification.
SECURITY_IMPERSONATION	Precizează că fişierul va personifica clientul la nivelul Impersonation.
SECURITY_DELEGATION	Precizează că fişierul va personifica clientul la nivelul Delegation.

(continuare)

Valoare	Semnificație
SECURITY_CONTEXT_TRACKING	Precizează că modul de urmărire a securității este dinamic. Dacă acest indicator nu este specificat, modul de urmărire al securității este static.
SECURITY_EFFECTIVE_ONLY	Precizează că numai atributele activate ale contextului de securitate al clientului sunt disponibile pentru server. Dacă nu menționați acest indicator, toate atributele contextului de securitate ale clientului sunt disponibile. Acest indicator permite clientului să limiteze grupurile și privilegiile utilizate de un server în timpul personificării clientului.

Tabelul 1453.1 *Indicatoarele de securitate utilizate la crearea unui canal de transfer nominal.*

Dacă funcția *CreateFile* deschide extremitatea client al unui slot pentru mail, funcția returnează *INVALID_HANDLE_VALUE* când clientul încearcă să deschidă un slot pentru mail local, înainte ca serverul slotului de mail să îl fi creat cu funcția *CreateMailSlot*.

Funcția *CreateFile* poate crea un identificator pentru o resursă de comunicații, cum ar fi portul serial COM1. Pentru resursele de comunicații, parametrul *dwCreationDistribution* trebuie să fie *OPEN_EXISTING*, iar parametrul *hTemplate* trebuie să fie NULL. Pot fi specificate modurile de acces de citire, scriere sau citire-scriere, iar identificatorul poate fi deschis pentru operațiuni I/O suprapuse

Sub Windows NT puteți utiliza funcția *CreateFile* pentru deschiderea unei unități de disc sau a unei partiții pe o unitate de disc. Funcția returnează un identificator pentru unitatea de disc. Programul poate folosi ulterior acest identificator cu funcția *DeviceIOControl*. Pentru reușită, apelul trebuie să îndeplinească următoarele cerințe:

- Apelantul trebuie să aibă privilegii de administrator, pentru ca operația să reușească pe unitatea de hard-disc.
- Șirul *lpFileName* trebuie să fie de forma \\.\ *physicaldrivex* pentru deschiderea unui hard-disc X. De exemplu, șirul \\.\ *physicaldrive2* obține identificatorul pentru a treia unitate fizică a calculatorului utilizat.
- Șirul *lpFileName* trebuie să fie \\.*x:* pentru a deschide o unitate de disc flexibil *x* sau o partiție *x* de pe hard-disc. De exemplu, șirul \\.*A:* obține un identificator pentru unitatea A de pe calculatorul utilizat, iar șirul \\.*C:* obține identificatorul pentru unitatea C a calculatorului respectiv.
- Sub Windows 95, această tehnică nu funcționează pentru deschiderea unei unități logice. În Windows 95, precizarea în *CreateFile* a unui șir de această formă are ca efect returnarea unei erori.
- Parametrul *dwCreationDistribution* trebuie să ia valoarea *OPEN_EXISTING*.
- La deschiderea unui disc flexibil sau a unei partiții pe hard-disc, în parametrul *dwShareMode* trebuie să stabiliți indicatorul *FILE_SHARE_WRITE*.

Funcția *CreateFile* poate crea un identificator pentru intrarea consolă (*CONIN$*). Dacă procesul are un identificator deschis către intrarea consolă ca rezultat al moștenirii sau duplicării, el poate de asemenea crea un identificator pentru bufferul activ al ecranului (*CONOUT$*). Trebuie să atașați procesul apelant la consola moștenită sau la una alocată prin

funcția *AllocConsole*. Pentru identificatorii de consolă, parametrii funcției *CreateFile* trebuie stabiliți în modul descris în Tabelul 1453.2.

Parametru	Valoare
lpFileName	Utilizați valoarea *CONIN$* pentru specificarea intrării la consolă și valoarea *CONOUT$* pentru specificarea ieșirii la consolă. *CONIN$* obține un identificator pentru bufferul de intrare al consolei, chiar dacă funcția *SetStdHandle* a redirecționat identificatorul standard de intrare. Pentru obținerea identificatorului standard de intrare, utilizați funcția *GetStdHandle*. *CONOUT$* obține un identificator pentru bufferul de ecran activ, chiar dacă *SetStdHandle* a redirectat identificatorul standard de ieșire. Pentru a obține identificatorul standard de ieșire, utilizați funcția *GetStdHandle*.
dwDesiredAccess	Microsoft recomandă utilizarea valorilor *GENERIC_READ* I *GENERIC_WRITE*, dar și unul singur poate limita accesul.
dwShareMode	Dacă procesul apelant a moștenit consola sau dacă un proces copil trebuie să aibă acces la consolă, parametrul trebuie să fie *FILE_SHARE_READ* I *FILE_SHARE_WRITE*.
lpSecurityAttributes	Dacă doriți ca procesul copil să moștenească consola, membrul *bInheritHandle* al structurii *SECURITY_ATTRIBUTES* trebuie să fie TRUE.
dwCreationDistribution	Trebuie să specificați valoarea *OPEN_EXISTING* când utilizați *CreateFile* pentru deschiderea consolei.
dwFlagsAndAttributes	Ignorat.
hTemplateFile	Ignorat.

Tabelul 1453.2 *Valorile pentru parametrii funcției* ***CreateFile****, la crearea consolelor.*

Tabelul 1453.3 prezintă efectele diferitelor valori ale parametrului *dwDirectAccess* când *lpFileName* are valoarea CON.

Valori	Rezultat
GENERIC_READ	Deschide consola pentru intrare.
GENERIC_WRITE	Deschide consola pentru ieșire.
GENERIC_READ I GENERIC_WRITE	Provoacă eșuarea funcției *CreateFile*.

Tabelul 1453.3 *Efectele valorilor de acces la deschiderea unei console.*

Aplicațiile nu pot crea directoare cu *CreateFile*. În schimb, pentru crearea unui director, aplicațiile trebuie să apeleze funcțiile *CreateDirectory* sau *CreateDirectoryEx*. Însă, sub Windows NT, puteți obține un identificator pentru un director prin indicatorul *FILE_FLAG_BACKUP_SEMANTICS*. Un identificator pentru un director poate fi transmis unor funcții Win32, în locul unui identificator de fișier. Unele sisteme de fișiere, cum ar fi NTFS (Prescurtare pentru sistemul de fișiere specific sistemului de operare Windows NT - New Technology File System), acceptă comprimarea directoarelor și fișierelor individuale. Pe volumele formatate pentru acest sistem de fișiere, un director nou moștenește atributele de comprimare ale directorului părinte.

1454 UTILIZAREA IDENTIFICATORILOR DE FIȘIER



La fel ca în DOS și UNIX, sistemul Windows atribuie un identificator de fișier pentru fiecare fișier pe care programul îl deschide sau îl creează. Un identificator de fișier este un identificator unic utilizat de o aplicație în cadrul funcției care accesează un fișier. Identificatorii de fișier sunt valizi până când aplicațiile îi închid cu funcția *CloseHandle*, care închide fișierul și eliberează bufferele fișierului prin scrierea conținutului său pe disc. În momentul primei lansări a unei aplicații, ea moștenește toți identificatorii de deschidere a fișierelor de la procesul care a lansat aplicația, cu condiția ca procesul părinte să fi deschis fișierele și să fi permis moștenirea. Dacă procesul părinte a deschis fișierele fără să fi permis moștenirea, aplicația nu va moșteni identificatorii de deschidere ai fișierelor.

O aplicație poate deschide identificatori de fișier pentru intrări și ieșiri la consolă. În locul numelui de fișier aplicația transmite șirul *CONIN$* funcției *CreateFile* ca nume de fișier pentru consola de intrare și *CONOUT$* ca nume de fișier pentru consola de ieșire.

1455 DIN NOU DESPRE POINTERII DE FIȘIER



Așa cum ați învățat în capitolul despre fișiere, directoare și discuri din această carte, când aplicația dumneavoastră deschide pentru prima dată un fișier, sistemele de operare plasează de regulă un pointer de fișier la începutul fișierului. Windows nu face excepție. Pointerul de fișier marchează poziția curentă în fișier, în locul unde va avea loc o operație de citire sau scriere. Pe măsură ce programul citește sau scrie fiecare octet din fișier, Windows avansează pointerul de fișier la următorul octet. O aplicație poate, de asemenea, să deplaseze poziția pointerului într-un fișier cu funcția *SetFilePointer*. Veți implementa funcția *SetFilePointer* potrivit prototipului prezentat mai jos:

```
DWORD SetFilePointer(
    HANDLE hFile,              // identificator de fisier
    LONG lDistanceToMove,      // numar de octeti cu care sa
                               // se mute pointerul
    PLONG lpDistanceToMoveHigh, // adresa cuvant mai
                               // semnificativ pt distanta de
                               // deplasare
    DWORD dwMoveMethod         // mod deplasare
);
```

Funcția *SetFilePointer* utilizează parametrii prezentați în Tabelul 1455.1.

Parametru	Descriere
hFile	Identifică fișierul al cărui pointer de fișier va fi deplasat. Identificatorul de fișier trebuie să fi fost creat cu accesul la fișier *GENERIC_READ* sau *GENERIC_WRITE*.
lDistanceToMove	Specifică numărul de octeți cu care va fi deplasat pointerul de fișier. O valoare pozitivă deplasează pointerul înainte, iar o valoare negativă, înapoi.

Parametru	Descriere
lpDistanceToMoveHigh	Indică cuvântul mai semnificativ al distanței pe 64 de biți pe care se face deplasarea. Dacă valoarea este NULL, *SetFilePointer* poate opera numai pe fișierele a căror dimensiune maximă este (2^{32} – 2). Dacă acest parametru este specificat, mărimea maximă a fișierului este (2^{64} – 2). Acest parametru primește de asemenea cuvântul mai semnificativ al noii valori a pointerului de fișier.
dwMoveMethod	Specifică punctul de pornire al deplasării pointerului de fișier. Acest parametru poate lua una din valorile descrise în Tabelul 1455.2.

Tabelul 1455.1 *Parametrii funcției* ***SetFilePointer***.

Există mai multe metode în Windows prin care puteți să deplasați pointerul de fișier. Tabelul 1455.2 enumeră metodele predefinite de deplasare a pointerului de fișier.

Valoare	Semnificație
FILE_BEGIN	Punctul de pornire este zero sau începutul fișierului. Dacă este specificat *FILE_BEGIN*, funcția interpretează *DistanceToMove* ca o locație fără semn pentru noul pointer de fișier.
FILE_CURRENT	Punctul de pornire îl reprezintă valoarea curentă a pointerului de fișier.
FILE_END	Punctul de pornire îl reprezintă poziția curentă de sfârșit de fișier.

Tabelul 1455.2 *Metodele posibile de deplasare pentru* ***SetFilePointer***.

Dacă funcția *SetFilePointer* reușește, ea returnează cuvântul dublu mai puțin semnificativ al noului pointer de fișier. Dacă *lpDistanceToMoveHigh* nu este NULL, funcția pune cuvântul dublu mai semnificativ al valorii noului pointer de fișier în valoarea *long* indicată de acest parametru. Dacă funcția eșuează și parametrul *lpDistanceToMoveHigh* nu este NULL, valoarea returnată este 0xFFFFFFFF, iar *GetLastError* va returna o altă valoare decât *NO_ERROR*.

Nu puteți utiliza funcția *SetFilePointer* cu un identificator de dispozitiv care nu efectuează căutări, cum ar fi dispozitivul canal de transfer sau un dispozitiv de comunicare. Pentru stabilirea tipului de fișier pentru *hFile*, utilizați funcția *GetFileType*. Trebuie să fiți prudent când fixați un pointer de fișier într-o aplicație multifir. O aplicație ale cărei fire partajează un identificator de fișier, actualizează pointerul de fișier și citește din fișier, trebuie să utilizeze un obiect secțiune critică sau excludere reciprocă (mutex) pentru protejarea actualizărilor pointerului de fișier. Dacă identificatorul de fișier *hFile* a fost deschis cu indicatorul *FILE_FLAG_NO_BUFFERING*, aplicația poate deplasa pointerul numai la pozițiile aliniate la sector. O poziție aliniată la sector reprezintă o poziție multiplu întreg al dimensiunii sectorului de volum. Aplicația poate obține dimensiunea sectorului de volum apelând funcția *GetDiskFreeSpace*. Dacă o aplicație apelează *SetFilePointer* cu valorile distanței de deplasat care rezultă într-o poziție nealiniată la sector și cu un identificator care a fost deschis inițial cu *FILE_FLAG_NO_BUFFERING*, funcția eșuează, iar *GetLastError* returnează *ERROR_INVALID_PARAMETER*.

Observație: SetFilePointer *este asemănătoare funcției* ***lseek***, *prezentată în secțiunea 408, și funcției* ***fseek***, *prezentată în secțiunea 450.*

1456 UTILIZAREA FUNCȚIEI WRITEFILE PENTRU SCRIEREA ÎN FIȘIER

C/C++

Programele dumneavoastră pot deschide fișierele cu funcția *CreateFile*. Când creați un fișier cu acces la scriere, programul poate apoi utiliza pentru scriere funcția *WriteFile*. Funcția *WriteFile* scrie date într-un fișier având posibilități de operare sincrone și asincrone. Funcția începe scrierea datelor la punctul indicat de pointerul de fișier. După ce operația de scriere a fost terminată, pointerul de fișier este poziționat corespunzător numărului de octeți efectiv scriși, exceptând situația când fișierul a fost deschis cu *FILE_FLAG_OVERLAPPED*. Dacă identificatorul de fișier a fost creat pentru operații de intrare/ieșire cu suprapunere, aplicația trebuie să schimbe poziția pointerului de fișier după încheierea operației de scriere. Veți implementa funcția *WriteFile* potrivit prototipului prezentat mai jos:

```
BOOL WriteFile(
    HANDLE hFile,      // identificator de fisier
    LPCVOID lpBuffer, // pointer la datele pentru scriere
    DWORD nNumberOfBytesToWrite, // numar de octeti de scris
    LPDWORD lpNumberOfBytesWritten, // pointer la numarul de
                                    // octeti scrisi
    LPOVERLAPPED lpOverlapped // pointer la structura pt. I/O
                              // suprapuse
);
```

Funcția *WriteFile* acceptă parametrii prezentați în Tabelul 1456.

Parametru	Semnificație
hFile	Identifică fișierul în care se scrie. Identificatorul de fișier trebuie să fi fost creat cu accesul de tip *GENERIC_WRITE* la fișier. Sub Windows NT, pentru operațiile de scriere asincrone, *hFile* poate fi orice identificator deschis cu valoarea *FILE_FLAG_OVERLAPPED* în funcția *CreateFile* sau un identificator de soclu returnat de funcțiile *socket* sau de *accept*. Sub Windows 95, pentru operațiile de scriere asincrone, *hFile* poate fi un identificator pentru o resursă de comunicații, de slot pentru mail sau de canal de transfer nominal deschis de *CreateFile* cu valoarea *FILEFLAG_OVERLAPPED* sau un identificator de soclu returnat de funcțiile *socket* sau *accept*. Windows 95 nu acceptă operații de scriere asincrone pe disc.
lpBuffer	Indică bufferul care conține datele care trebuie scrise în fișier.
nNumberOfBytesToWrite	Specifică numărul de octeți de scris în fișier. Spre deosebire de sistemul de operare MS-DOS, Windows NT interpretează valoarea zero pentru specificarea unei operații de scriere nulă. O operație de scriere nulă nu scrie nici un octet, dar determină modificări la marcarea timpului.

Parametru	Semnificaţie
lpNumberOfBytesWritten	Indică numărul de octeţi scrişi de apelul la funcţia *WriteFile*. Această funcţie fixează valoarea parametrului la zero, înaintea oricărei acţiuni sau testări de eroare. Dacă *lpOverlapped* este *NULL*, *lpNumberOfBytesWritten* nu poate fi *NULL*. Dacă *lpOverlapped* nu este *NULL*, *lpNumberOfBytesWritten* poate fi NULL. Dacă aveţi o operaţie de scriere cu suprapunere, puteţi obţine numărul de octeţi scrişi prin apelarea funcţiei *GetOverlappedResult*. Dacă *hFile* este asociat cu un port de completare I/O, puteţi obţine numărul de octeţi scrişi apelând *GetQueuedCompletionStatus*.
lpOverlapped	Indică o structură *OVERLAPPED*. Această structură este cerută dacă *hFile* a fost deschis cu *FILE_FLAG_OVERLAPPED*, iar parametrul *lpOverlapped* trebuie să fie în acest caz NULL. El trebuie să indice o structură *OVERLAPPED* validă. Dacă *hFile* a fost deschis cu *FILE_FLAG_OVERLAPPED*, iar *lpOverlapped* este NULL, funcţia poate raporta incorect că operaţia de scriere este completă. Dacă *hFile* a fost deschis cu *FILE_FLAG_OVERLAPPED*, iar *lpOverlapped* nu este NULL, operaţia de scriere începe la deplasamentul specificat în structura *OVERLAPPED*, iar *WriteFile* se poate returna înaintea încheierii operaţiei de scriere. În acest caz *WriteFile* returnează FALSE, iar funcţia *GetLastError* returnează *ERROR_IO_PENDING*. Folosirea operaţiilor de intrare/ieşire cu suprapunere permite procesului apelant să continue prelucrarea în timp ce sistemul de operare încheie operaţia de scriere. Evenimentul specificat în structura *OVERLAPPED* este fixat pe starea semnalizat la încheierea operaţiei de scriere. Dacă *hFile* nu este deschis cu *FILE_FLAG_OVERLAPPED* şi *lpOverlapped* este *NULL*, operaţia de scriere începe la poziţia curentă a fişierului, iar *WriteFile* nu se returnează decât în momentul încheierii operaţiei de scriere. Dacă *hFile* nu a fost deschis cu *FILE_FLAG_OVERLAPPED* şi *lpOverlapped* nu este NULL, operaţia de scriere începe la deplasamentul specificat de structura *OVERLAPPED*, iar *WriteFile* nu se returnează decât în momentul încheierii operaţiei de scriere.

Tabelul 1456 *Parametrii funcţiei* ***WriteFile***.

Dacă funcţia *WriteFile* reuşeşte, valoarea returnată este diferită de zero. Dacă funcţia eşuează, valoarea returnată este zero. Dacă o parte a fişierului este blocată de un alt proces, iar operaţia de scriere se suprapune cu porţiunea blocată, această funcţie eşuează. Aplicaţiile nu trebuie să citească sau să scrie într-un buffer de ieşire folosit de operaţia de scriere, decât după încheierea operaţiei de scriere. Accesul prematur la bufferul de ieşire poate duce la coruperea datelor scrise din acel buffer.

Programul poate folosi *WriteFile* cu un identificator pentru ieşirea consolă pentru scrierea caracterelor la un buffer de ecran. Comportamentul precis al funcţiei este determinat de modul consolă. Datele sunt scrise la poziţia curentă a cursorului. Poziţia cursorului este actualizată de sistemul de operare după operaţia de scriere. Spre deosebire de sistemul de operare MS-DOS, sistemul Windows NT interpretează zero octeţi scrişi drept o operaţie de

scriere nulă, iar *WriteFile* nu trunchiază sau extinde fişierul. Pentru trunchierea sau extinderea fişierului, utilizaţi funcţia *SetEndOfFile*.

Când o aplicaţie utilizează funcţia *WriteFile* pentru scrierea la un canal de transfer, operaţia de scriere ar putea să nu se termine dacă bufferul canalului de transfer este plin. Operaţia de scriere este încheiată când operaţia de citire (utilizând funcţia *ReadFile*) creează spaţiu disponibil în buffer. Dacă identificatorul pentru canalul de transfer anonim a fost închis, iar *WriteFile* încearcă să utilizeze identificatorul corespunzător canalului de transfer anonim, funcţia returnează valoarea FALSE, iar *GetLastError* returnează *ERROR_BROKEN_PIPE*.

Funcţia *WriteFile* poate eşua returnând *ERROR_INVALID_USER_BUFFER* sau *ERROR_NOT_ENOUGH_MEMORY* de fiecare dată când există prea multe cereri asincrone de I/O. Pentru anularea operaţiilor de I/O asincrone în aşteptare, utilizaţi funcţia *CancelIO*. Această funcţie va anula doar operaţiile ce provin din firul apelant pentru identificatorul de fişier specificat. Operaţiile de I/O anulate ca rezultat al apelării funcţiei *CancelIO* sunt încheiate cu eroarea *ERROR_OPERATION_ABORTED*.

Dacă încercaţi să scrieţi la o unitate de disc flexibil care nu conţine un disc, sistemul afişează o casetă de mesaj cerând utilizatorului să reîncerce operaţia. Pentru a preveni ca sistemul să afişeze caseta de mesaj, apelaţi funcţia *SetErrorMode* cu *SEM_NOOPENFILEERRORBOX*. Dacă *hFile* este un identificator pentru un canal de transfer nominal, membrii *Offset* şi *OffsetHigh* ai structurii *OVERLAPPED*, indicaţi de *lpOverlapped*, trebuie să fie zero, altfel funcţia eşuează.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Write_File.cpp* care deschide fişierul *file.dat* şi scrie un şir în acest fişier în momentul când programul de demonstraţie porneşte. Când utilizatorul selectează opţiunea *Test!*, programul va scrie o altă linie de text în fişier şi apoi va afişa ambele linii de text într-o casetă de mesaj.

1457 UTILIZAREA FUNCŢIEI READFILE PENTRU CITIREA DIN FIŞIER



Aşa cum aţi învăţat în secţiunea 1456, programul dumneavoastră poate utiliza funcţia *WriteFile* pentru scrierea într-un fişier în Windows. În mod asemănător, programele pot utiliza funcţia *ReadFile* pentru citirea dintr-un fişier. Funcţia *ReadFile* citeşte date dintr-un fişier, începând cu poziţia indicată de pointerul de fişier. După ce operaţia de citire este completă, pointerul de fişier este poziţionat corespunzător numărului de octeţi citiţi efectiv, cu excepţia situaţiei când identificatorul de fişier a fost creat cu atributul de suprapunere. Dacă identificatorul de fişier este creat pentru intrări şi ieşiri suprapunere, aplicaţia trebuie să refacă poziţia pointerului de fişier după operaţia de citire. Veţi implementa funcţia *ReadFile* potrivit prototipului prezentat mai jos:

```
BOOL ReadFile(
    HANDLE hFile,                 // identificator de fisier
    LPVOID lpBuffer,              // adresa din bufferul care
                                  // primeste datele
    DWORD nNumberOfBytesToRead,   // numar de octeti de citit
    LPDWORD lpNumberOfBytesRead,  // adresa cu numar de octeti
                                  // de citit
    LPOVERLAPPED lpOverlapped     // adresa structurii pt. date
);
```

Funcția *ReadFile* acceptă aceiași parametri cu funcția *WriteFile* prezentați în secțiunea 145(cu excepția faptului că funcția *ReadFile* așteaptă parametrul *nNumberOfBytesToRead* î locul parametrului *nNumberOfBytesToWrite* și returnează parametrul *lpNumberOfBytesRea* și nu *lpNumberOfBytesWritten*. Parametrul *nNumberOfBytesToRead* specifică numărul d octeți citiți dintr-un fișier. Parametrul *lpNumberOfBytesRead* indică numărul de octeți citi *ReadFile* stabilește această valoare la zero înaintea oricărei operații sau testări. Dac *lpNumberOfBytesRead* este zero când *ReadFile* returnează TRUE de la un canal de transf nominal, celălalt capăt al canalului de transfer în mod mesaj a apelat funcția *WriteFile* c *nNumberOfBytesToWrite* la valoarea zero.

Dacă *lpOverlapped* este NULL, *lpNumberOfBytesRead* nu poate fi NULL. Dacă *lpOverlappe* nu este NULL, *lpNumberOfBytesRead* poate fi NULL. Dacă este o operație de citire cu supra punere, puteți obține numărul de octeți citiți prin apelarea funcției *GetOverlappedResul* Dacă *hFile* este asociat cu un port de completare I/O, puteți obține numărul de octeți citi apelând *GetQueuedCompletionStatus*.

Dacă *hFile* a fost deschis cu *FILE_FLAG_OVERLAPPED*, parametrul *lpOverlapped* nu trebui să fie NULL. El trebuie să indice o structură *OVERLAPPED* validă. Dacă *hFile* a fost deschis c *FILE_FLAG_OVERLAPPED*, iar *lpOverlapped* este NULL, funcția poate raporta incorect c operația de citire este completă. Dacă *hFile* a fost deschis cu *FILE_FLAG_OVERLAPPED*, ia *lpOverlapped* nu este NULL, operația de citire începe la deplasamentul specificat în structur *OVERLAPPED*, iar *ReadFile* se poate returna înaintea încheierii operației de citire. În aces caz, *ReadFile* returnează FALSE, iar funcția *GetLastError* returnează *ERROR_IO_PENDING* Aceasta permite procesului apelant să continue prelucrarea pe parcursul completării operației de citire. Evenimentul specificat în structura *OVERLAPPED* este fixat pe starea semnalizat, indicând încheierea operației de citire.

Dacă *hFile* nu este deschis cu *FILE_FLAG_OVERLAPPED* și *lpOverlapped* este *NULL*, operația de citire începe la poziția curentă a fișierului, iar *ReadFile* nu se returnează decât în momentul încheierii operației de citire. Dacă *hFile* nu a fost deschis cu *FILE_FLAG_OVERLAPPED* și *lpOverlapped* nu este NULL, operația de citire începe la deplasamentul specificat de structura *OVERLAPPED*, iar *ReadFile* nu se returnează decât în momentul încheierii operației de citire.

Dacă funcția reușește, valoarea returnată este diferită de zero. Dacă valoarea returnată este diferită de zero, iar numărul de octeți citit este zero, pointerul de fișier a fost dincolo de sfârșitul curent al fișierului la momentul operației de citire. Însă, dacă fișierul a fost deschis cu *FILE_FLAG_OVERLAPPED*, iar *lpOverlapped* nu este NULL, valoarea returnată este FALSE și *GetLastError* returnează *ERROR_HANDLE_EOF* când pointerul de fișier trece dincolo de sfârșitul curent al fișierului. Dacă funcția eșuează, valoarea returnată este zero.

ReadFile se returnează când are loc unul din următoarele evenimente: o operație de scriere se încheie la extremitatea de scriere a unui canal de transfer, numărul de octeți cerut a fost citit sau a apărut o eroare. Dacă o parte a fișierului a fost blocată de un alt proces și operația de citire se suprapune pe porțiunea blocată, funcția eșuează. Aplicațiile nu trebuie să citească sau să scrie într-un buffer de intrare folosit de operația de citire, decât după încheierea operației de citire. Accesul prematur la bufferul de intrare poate duce la coruperea datelor citite în buffer. Programul poate folosi funcția *ReadFile* cu un identificator de intrare consolă pentru citirea caracterelor la un buffer de intrare. Comportamentul precis al funcției *ReadFile* este determinat de modul consolă.

Când o aplicație utilizează funcția *ReadFile* pentru citirea de la un canal de transfer în modul mesaj și următorul mesaj este mai lung decât se precizează în parametrul

nNumberOfBytesToRead, *ReadFile* returnează FALSE și *GetLastError* returnează *ERROR_MORE_DATA*. Restul mesajului poate fi citit printr-un apel ulterior la funcțiile *ReadFile* sau *PeekNamedPipe*. Când citiți de pe un dispozitiv de comunicații, comportamentul funcției *ReadFile* este dirijat de întreruperile curente ale comunicației, stabilite sau obținute folosind funcțiile *SetCommTimeouts* și *GetCommTimeouts*. Pot apărea rezultate neașteptate dacă nu fixați valorile de întrerupere. Dacă *ReadFile* încearcă să citească de pe un slot pentru mail al cărui buffer este prea redus, funcția returnează FALSE, iar *GetLastError* returnează *ERROR_INSUFFICIENT_BUFFER*.

Dacă procesul a închis un identificator de un canal de transfer anonim și *ReadFile* încearcă să citească utilizând identificatorul corespunzător la canalul de transfer anonim de citire, funcția returnează FALSE, iar *GetLastError* returnează *ERROR_BROKEN_PIPE*. Funcția *ReadFile* poate eșua returnând *ERROR_INVALID_USER_BUFFER* sau *ERROR_NOT_ENOUGH_MEMORY* de fiecare dată când există prea multe cereri asincrone de I/O în așteptare. Programul *Write_File.cpp* prezentat în secțiunea 1456, demonstrează și utilizarea funcției *ReadFile*.

Observație: *Codul* ***ReadFile*** *pentru testarea condiției de sfârșit de fișier (eof) diferă în cazul operațiunilor de citire sincrone și asincrone. Când o operație de citire sincronă atinge sfârșitul fișierului,* ***ReadFile*** *returnează TRUE și fixează la zero* ****lpNumberOfBytesRead***. *Când o operație de citire asincronă atinge sfârșitul fișierului,* ***ReadFile*** *eșuează și returnează* ***ERROR_HANDLE_EOF***.

1458 ÎNCHIDEREA FIȘIERULUI



Identificatorii de fișiere se utilizează pentru operațiile cu fișiere în programele Windows. Ca și în cazul altor identificatori (de memorie sau pentru contextele de dispozitiv) trebuie întotdeauna să închideți un identificator imediat ce programul a încheiat prelucrarea sa. Funcția *CloseHandle* închide un identificator deschis pentru un obiect. Veți implementa funcția *CloseHandle* potrivit prototipului prezentat mai jos:

```
BOOL CloseHandle( HANDLE hObject );
```

Parametrul *hObject* identifică un identificator de obiect deschis. Dacă funcția reușește valoarea returnată este diferită de zero, iar dacă eșuează valoarea returnată este zero. Funcția *CloseHandle* închide identificatorii următoarelor obiecte:

- Intrări sau ieșiri la consolă
- Fișier eveniment
- Mapare de fișier
- Excludere reciprocă (mutex)
- Canal de transfer nominal
- Proces
- Semafor
- Fir
- Token (numai în Windows NT)

Funcția *CloseHandle* invalidează identificatorul respectiv de obiect, decrementează valoarea de contor a identificatorului de obiect și efectuează testele de retenție ale obiectului. După ce procesul a închis ultimul identificator de obiect, obiectul este eliminat de sistemul de operare din prelucrările sale. Funcția *CloseHandle* nu închide obiectele modul. Închiderea unui identificator nevalid creează o eroare. Aceasta implică și închiderea de două ori a identificatorului, netestarea valorii returnate și închiderea unui identificator nevalid, precum și utilizarea funcției *CloseHandle* pentru un identificator returnat de *FindFirstFile*.

Observație: *Utilizați* ***CloseHandle*** *pentru închiderea identificatorilor returnați de funcția* ***CreateFile****. Utilizați* ***FindClose*** *pentru închiderea identificatorilor returnați de funcția* ***FindFirstFile****.*

PARTAJAREA DATELOR PRIN MAPAREA FIȘIERELOR

C/C++1459

Maparea fișierului reprezintă copierea conținutului fișierului în spațiul de adresă virtual al procesului. Când mapați un fișier, copia conținutului fișierului este cunoscută cu denumirea de *vizualizare de fișier* (*file view*), iar structura internă a folosită de program pentru menținerea copiei este cunoscută cu denumirea de *obiect de mapare fișier*. Pentru partajarea datelor, un alt proces poate utiliza obiectul de mapare fișier al primului proces în vederea producerii unei vizualizări de fișier identice în propriul său spațiu de adresă virtual.

Un exemplu obișnuit de date partajate între procese este schimbul de date dinamice (DDE) și „fratele său mai mic" OLE. În Windows 3.x, aplicațiile alocă memorie globală cu indicatorul *GMEM_DDESHARE* și utilizează identificatorul de memorie pentru partajarea datelor între procese. În Windows 95 și Windows NT, aplicațiile utilizează în schimb maparea fișierelor. Pentru aplicațiile dumneavoastră nu este nevoie efectiv de maparea unui fișier în memorie. O aplicație poate specifica valoarea *0xFFFFFFFF* pentru identificatorul de fișier în momentul apelării funcției *CreateFileMapping* care va mapa o vizualizare a fișierului de paginare (despre care ați învățat anterior) a sistemului de operare, în memorie. Secțiunea 1460 va explica în detaliu funcția *CreateFileMapping*.

În momentul mapării unui fișier în memorie, programul poate accesa în întregime conținutul fișierului ca și cum fișierul ar fi o matrice. Programul poate utiliza chiar valori de index și pointeri pentru accesarea conținutului fișierelor.

MAPAREA UNUI FIȘIER ÎN MEMORIA VIRTUALĂ

C/C++1460

Așa cum ați învățat în secțiunea 1459, programul va efectua frecvent mapări de fișiere în spațiul de adresă virtual a unui proces. Maparea fișierelor va facilita accesul programului la datele din fișiere, mai rapid și eficient. Când doriți să mapați un fișier în memorie, programul va trebui să utilizeze funcția *CreateFileMapping* pentru crearea unui obiect de mapare fișier nominal sau fără nume, pentru fișierul specificat. Veți implementa funcția *CreateFileMapping* potrivit prototipului prezentat mai jos:

```
HANDLE CreateFileMapping(
    HANDLE hFile, // identificator de fisier pt. mapare
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes, //atribute
```

```
                                        // optionale de
                                        // securitate
    DWORD flProtect,            // protectie pt. obiectul de mapare
    DWORD dwMaximumSizeHigh,    // 32 biti mai semnificativi din
                                // marimea obiectului
    DWORD dwMaximumSizeLow,     // 32 biti mai putin semnificativi
                                // din marimea obiectului
    LPCTSTR lpName // numele obiectului de mapare fisier
);
```

Funcția *CreateFileMapping* acceptă parametrii prezentați în Tabelul 1460.1.

Parametru	Descriere
hFile	Identifică fișierul de la care se va crea obiectul de mapare. Fișierul trebuie să fie deschis cu un mod de acces compatibil cu indicatoarele de protecție specificate de parametrul *flProtect*. Microsoft recomandă, deși Windows nu necesită acest lucru, ca fișierele pe care intenționați să le mapați să fie deschise cu acces exclusiv. Dacă *hFile* este (*HANDLE*)0xFFFFFFFF, procesul apelant trebuie, de asemenea, să precizeze dimensiunea obiectului de mapare în parametrii *dwMaximumSizeHigh* și *dwMaximumSizeLow*. Funcția creează un obiect de mapare fișier cu dimensiunea specificată, susținut de fișierul de paginare al sistemului de operare și nu de un fișier cu nume din sistemul de fișiere. Obiectul de mapare fișier poate fi partajat prin duplicare, prin moștenire sau prin nume.
lpFileMappingAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care stabilește dacă identificatorul returnat poate fi moștenit de procesele copil. Dacă *lpFileMappingAttributes* este NULL, identificatorul nu poate fi moștenit de procesele copil.
flProtect	Specifică protecția dorită pentru vizualizarea fișierului la maparea fișierului. *PAGE_READONLY* asigură acces la citire (*read-only*) în regiunea de pagini angajată. O încercare de a scrie sau executa o regiune angajată, duce la o violare de acces. Fișierele specificate de parametrul *hFile* trebuie să fi fost create cu drepturile de acces *GENERIC_READ*. PAGE_READWRITE asigură acces citire-scriere în regiunea de pagini angajată. Fișierele specificate de parametrul *hFile* trebuie să fi fost create cu drepturile de acces *GENERIC_READ* și *GENERIC_WRITE*. PAGE_WRITECOPY conferă acces prin copiere la scriere în regiunea de pagini angajată. Fișierele specificate *de* parametrul *hFile* trebuie să fi fost create cu drepturile de acces *GENERIC_READ* și *GENERIC_WRITE*. În plus, o aplicație poate combina unul sau mai multe atribute de secțiune (prin operatorul OR pe bit) prezentate în Tabelul 1460.2 cu una din valorile de protecție prezentate în urmă cu câteva pagini pentru a specifica anumite atribute de secțiune.

Parametru	Descriere
dwMaximumSizeHigh	Specifică cei 32 de biți mai semnificativi ai dimensiunii maxime a obiectului de mapare fișier.
dwMaximumSizeLow	Specifică cei 32 de biți mai puțin semnificativi ai dimensiunii maxime a obiectului de mapare fișier. Dacă acest parametru și *dwMaximumSizeHig* sunt zero, dimensiunea maximă a obiectului de mapare fișier este egală cu dimensiunea curentă a fișierului identificat de *hFile*.
lpName	Indică un șir terminat cu NULL care specifică numele obiectului de mapare. Numele poate conține orice caracter cu excepția lui *backslash* (\). Dacă parametrul corespunde numelui unui obiect de mapare existent, funcția cere acces la obiectul de mapare cu atributul de protecție specificat de *flProtect*. Dacă parametrul este NULL, obiectul de mapare este creat fără nume.

Tabelul 1460.1 *Parametrii funcției* ***CreateFileMapping***.

Așa cum ați aflat din Tabelul 1460.1, puteți combina valorile de protecție ale fișierului cu unul sau mai multe atribute de secțiune. Tabelul 1460.2 descrie valorile de protecție ale fișierului de mapare a memoriei.

Valoare	Descriere
SEC_COMMIT	Alocă stocare fizică în memorie sau în fișierul de paginare pe disc, pentru toate paginile din secțiune. Este valoarea implicită.
SEC_IMAGE	Fișierul specificat pentru maparea fișierelor din secțiune este un fișier imagine executabil. Deoarece informația de mapare și protecția de fișier sunt luate din fișierul imagine, nici un alt atribut nu e valid cu *SEC_IMAGE*.
SEC_NOCACHE	Toate paginile unei secțiuni sunt stabilite fără stocare în memoria *cache*. Pe mașinile 80x86 și MIPS utilizarea memoriei *cache* pentru aceste structuri reduce performanța deoarece hardware-ul menține memoria *cache* coerentă. Unele drivere de dispozitiv cer date fără memorare în *cache* astfel încât programele să poată scrie prin memoria fizică. *SEC_NOCACHE* cere și valoarea *SEC_RESERVE* sau *SEC_COMMIT*.
SEC_RESERVE	Rezervă toate paginile unei secțiuni fără alocare de memorie fizică. Intervalul rezervat de pagini nu poate fi utilizat de alte operații de alocare până când nu este eliberat. Paginile rezervate pot fi angajate în apelurile ulterioare la funcția *VirtualAlloc*. Atributul *SEC_RESERVE* este valid numai dacă parametrul *hFile* este *(HANDLE)0xFFFFFFFF*; adică un obiect de mapare fișier susținut de fișierul de paginare al sistemului de operare.

Tabelul 1460.2 *Valorile de protecție de pagină ale fișierului de mapare a memoriei.*

Dacă funcția reușește, valoarea returnată este un identificator pentru obiectul de mapare fișier. Dacă obiectul de mapare a existat înaintea apelării funcției, *GetLastError* returnează *ERROR_ALREADY_EXISTS*, iar valoarea returnată este un identificator valid la obiectul existent de mapare fișier (cu dimensiunea sa curentă, nu cea nou specificată). Dacă obiectul de mapare nu există, *GetLastError* returnează zero. Dacă funcția eșuează, valoarea returnată este zero.

După ce un obiect de mapare fișier a fost creat, dimensiunea fișierului trebuie să nu depășească dimensiunea obiectului de mapare fișier. Dacă o depășește, nu va fi disponibil pentru partajare întregul conținut al fișierului. Dacă o aplicație specifică o dimensiune pentru

obiectul de mapare fișier mai mare decât dimensiunea fișierului curent pe disc, fișierul de pe disc va fi mărit pentru a corespunde mărimii obiectului de mapare fișier. Identificatorul returnat de *CreateFileMapping* are acces deplin la noul obiect de mapare fișier. El poate fi utilizat cu orice funcție care cere un identificator la un obiect de mapare fișiere. Obiectele de mapare fișier pot fi partajate prin crearea procesului, prin duplicarea identificatorului sau prin nume.

***Observație:** Sub Windows 95, identificatorii de fișier utilizați la crearea obiectelor de mapare fișier nu trebuie să fie folosiți în apelurile ulterioare la funcțiile I/O, cum ar fi **ReadFile** și **WriteFile**. În general, dacă un identificator de fișier a fost utilizat într-un apel reușit la funcția **CreateFileMapping**, nu utilizați acest identificator până când nu ați închis mai întâi obiectul de mapare fișier corespunzător.*

Crearea unui obiect de mapare fișier creează o mapare potențială a vizualizării fișierului, dar nu mapează vizualizarea. Funcțiile *MapViewOfFile* și *MapViewOfFileEx* mapează vizualizarea unui fișier în spațiul de adresă al unui proces. Secțiunea 1461 va explica funcția *MapViewOfFile* în detaliu.

În afară de o importantă excepție, vizualizările de fișier derivate dintr-un singur obiect de mapare fișier sunt coerente sau identice la un moment dat. Dacă mai multe procese dețin identificatori ai aceluiași obiect de mapare fișier, ele văd o vizualizare coerentă a datelor când mapează o vizualizare a fișierului. Excepția se referă la fișierele de la distanță. Deși *CreateFileMapping* operează cu fișiere la distanță, ea nu poate să le mențină coerența. De exemplu, dacă două calculatoare mapează un fișier apt la scriere și ambele modifică aceeași pagină, calculatoarele vor vedea numai ceea ce fiecare a scris în pagină. Când datele se actualizează pe disc, ele nu sunt unificate. Mai mult, un fișier mapat și un fișier normal, accesate prin funcțiile I/O – *ReadFile* și *WriteFile* – nu sunt în mod necesar, coerente.

Pentru închiderea completă a unui obiect de mapare fișier, aplicația trebuie să demapeze toate vizualizările mapate ale obiectului de mapare fișier prin apelarea funcției *UnmapViewOfFile* și să închidă identificatorul obiectului de mapare fișier prin apelarea funcției *CloseHandle*. Ordinea în care aceste funcții sunt apelate nu are importanță. Apelarea lui *UnmapViewOfFile* este necesară deoarece vizualizările mapate ale unui obiect de mapare fișier mențin identificatorii interni deschiși la obiect, iar obiectul de mapare fișier nu se închide până nu sunt închiși toți identificatorii săi deschiși.

1461 *MAPAREA UNEI VIZUALIZĂRI DE FIȘIER ÎN CADRUL PROCESULUI CURENT*



Așa cum ați învățat în secțiunea 1460, după ce programele dumneavoastră creează obiecte de mapare fișier, trebuie apoi să mapeze fișierul în cadrul obiectului. Pentru aceasta, programele trebuie să utilizeze funcția *MapViewOfFile*. Această funcție mapează vizualizarea unui fișier în spațiul de adresă al procesului apelant. Veți implementa funcția *MapViewOfFile* potrivit prototipului prezentat mai jos:

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject, // obiectul de mapat in spatiul
                               // de adresa
    DWORD dwDesiredAccess,     // mod de acces
    DWORD dwFileOffsetHigh,    // 32 biti mai semnificativi ai
```

```
                              // deplasamentului
    DWORD dwFileOffsetLow,    // 32 biti mai putin semnificativi
                              // ai deplasamentului
    DWORD dwNumberOfBytesToMap // numar octeti pt. mapare
);
```

Parametrul *hFileMappingObject* este un identificator deschis la un obiect de mapare fișier. Funcțiile *CreateFileMapping* și *OpenFileMapping* returnează acest identificator. Parametrul *dwDesiredAccess* specifică tipul de acces la vizualizarea fișierului și, în consecință, modul de protecție al paginilor mapate de fișier. Acest parametru poate lua una din valorile descrise în Tabelul 1461.

Valoare	Semnificație
FILE_MAP_WRITE	Acces permis la citire-scriere. Parametrul *hFileMappingObject* trebuie să fi fost creat cu protecția *PAGE_READWRITE*. Sistemul de operare mapează vizualizarea fișierului cu modul citire-scriere.
FILE_MAP_READ	Acces permis numai la citire. Parametrul *hFileMappingObject* trebuie să fi fost creat cu protecția *PAGE_READWRITE* sau *PAGE_READONLY*. Sistemul de operare mapează vizualizarea fișierului cu modul citire (*read only*).
FILE_MAP_ALL_ACCESS	Similar cu *FILE_MAP_WRITE*.
FILE_MAP_COPY	Acces permis la copiere. Dacă ați creat o mapare cu *PAGE_WRITECOPY* și vizualizarea cu *FILE_MAP_COPY*, veți obține o vizualizare a fișierului. Dacă scrieți în această vizualizare, paginile pot fi automat substituite, iar modificările nu se vor produce în fișierul de date inițial. Sub Windows 95, trebuie să transmiteți *PAGE_WRITECOPY* către funcția *CreateFileMapping;* în caz contrar, va apărea o eroare. Sub Windows NT, nu există restricții în privința modului în care trebuie creat parametrul *hFileMappingObject*. Modurile de acces la copiere sau la scriere sunt valide pentru orice tip de vizualizare. Dacă partajați maparea între mai multe procese cu *DuplicateHandle* sau *OpenFileMapping*, iar un proces scrie într-o vizualizare, modificările sunt propagate la celălalt proces. Fișierul inițial nu se modifică.

Tabelul 1461 *Valorile posibile ale parametrului* ***dwDesiredAccess***.

Parametrul *dwFileOffsetHigh* specifică cei 32 de biți mai semnificativi ai deplasamentului de fișier de unde începe maparea. Parametrul *dwFileOffsetLow* specifică cei 32 de biți mai puțin semnificativi ai deplasamentului de fișier de unde începe maparea. Combinarea celor două valori de deplasament trebuie să precizeze un deplasament din cadrul fișierului, care corespunde granularității de alocare a memoriei sistemului, altfel funcția eșuează. Adică, deplasamentul trebuie să fie un multiplu al granularității de alocare (de pildă, 8 octeți sau 16 octeți). Parametrul *dwNumberOfBytesToMap* specifică numărul de octeți ai fișierului ce urmează a fi mapat. Dacă *dwNumberOfBytesToMap* este zero, este mapat întregul fișier.

Dacă funcția reușește, valoarea returnată este adresa de început a vizualizării mapate. Dacă funcția eșuează, valoarea returnată este NULL. Maparea unui fișier face vizibilă porțiunea de fișier specificată, în spațiul de adresă al procesului apelant.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Simple_Map.cpp*. Acest program creează un fişier mapat în memorie în momentul când se lansează aplicaţia. Când utilizatorul selectează opţiunea *Test!* din meniu, programul mapează o vizualizare a fişierului şi plasează datele în memorie. Apoi, programul creează un fir şi aşteaptă într-un *timer* (cronometru) ca firul să modifice datele. Firul foloseşte datele pentru a afişa o casetă de mesaje şi, când utilizatorul închide caseta, plasează şirul „Received!" în memoria mapată. Când programul primeşte mesajul *WM_TIMER*, va testa dacă firul a plasat şirul în memoria mapată. Dacă şirul a fost plasat, programul încheie firul şi modulul *timer* avertizând utilizatorul.

1462 DESCHIDEREA UNUI OBIECT DE MAPARE FIŞIER DENUMIT



Programele dumneavoastră vor utiliza funcţia *CreateFileMapping* pentru crearea unui fişier de mapare denumit sau anonim şi funcţia *MapViewOfFile* pentru maparea unui fişier în memorie. În programul *Simple_Map.cpp*, un al doilea fir deschide fişierul mapat. În loc să realizeze o nouă mapare de fişier, cel de-al doilea fir utilizează funcţia *OpenFileMapping* pentru deschiderea fişierului mapat pentru propria sa folosinţă. Funcţia *OpenFileMapping* deschide un obiect de mapare fişier denumit. Veţi implementa funcţia *OpenFileMapping* potrivit prototipului prezentat mai jos:

```
HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,  // mod acces
    BOOL bInheritHandle,    // identificator mostenit
    LPCTSTR lpName          // pointer la numele obiectului
                            // de mapare fisier
);
```

Parametrul *dwDesiredAccess* specifică modul de acces la obiectul de mapare fişier. Sub Windows NT, parametrul de acces este testat faţă de orice descriptor de securitate al obiectului ţintă de mapare fişier. Acest parametru trebuie să ia una din valorile descrise în Tabelul 1461. Parametrul *bInheritHandle* precizează dacă identificatorul returnat va fi moştenit de un nou proces în momentul producerii acestui proces. Valoarea *TRUE* indică faptul că noul proces moşteneşte identificatorul. Parametrul *lpName* indică un şir care denumeşte obiectul de mapare fişier ce va fi deschis. Dacă există un identificator deschis la obiectul de mapare fişier cu acest nume iar descriptorul de securitate al obiectului de mapare nu intră în contradicţie cu parametrul *dwDesiredAccess*, operaţia de deschidere are loc cu succes.

Dacă funcţia reuşeşte, valoarea returnată este un identificator deschis la obiectul de mapare fişier specificat. Dacă funcţia eşuează, valoarea returnată va fi NULL. Puteţi utiliza identificatorul returnat de *OpenFileMapping* cu orice funcţie care necesită un identificator pentru un obiect de mapare fişier.

1463 ATRIBUTELE DE FIŞIER



În capitolele despre fişiere, directoare şi discuri din această carte, aţi învăţat despre atributele de fişier pe care DOS le ataşează fiecărui fişier pe care-l produceţi. În mod asemănător, Windows asociază fiecărui fişier un set de atribute. Windows iniţializează multe din aceste atribute la crearea fişierului, apoi modifică unele din aceste atribute de fiecare dată când

accesați fișierul. Cel mai adesea, nu veți schimba un atribut de fișier, ci vă veți rezuma la citirea fișierelor reacționând corespunzător. Ca și în DOS, majoritatea atributelor se referă la valori de indicatoare, dimensiuni de fișier și marcări de timp. În următoarele secțiuni, veți manevra și accesa unele din cele mai utilizate atribute de fișier.

OBȚINEREA ȘI SCHIMBAREA ATRIBUTELOR DE FIȘIER

C/C++1464

Așa cum ați învățat în secțiunea 1463, Windows atașează atribute fiecărui fișier creat în cadrul sistemului de operare. Așa cum ați citit, atributele de fișier cu funcțiile C standard prezentate în capitolele despre fișiere, directoare și discuri, și interfața Windows API pune la dispoziție multe funcții de citire a atributelor de fișier. Funcția *GetFileAttribute* returnează informații dintr-un subset din toate atributele posibile ale unui anumit fișier sau director. În secțiunile următoare se vor prezenta și alte funcții care returnează alte atribute de fișier, cum ar fi data unui fișier sau dimensiunea lui. Veți implementa funcția *GetFileAttribute* potrivit prototipului prezentat mai jos:

```
DWORD GetFileAttributes( LPCTSTR lpFileName );
```

Parametrul *lpFileName* indică un șir terminat cu NULL care conține numele fișierului sau directorului. Sub Windows NT, mărimea șirului cu numele de cale se limitează la o valoare implicită de *MAX_PATH* caractere. Această limită rezultă din modalitatea în care funcția *GetFileAttributes* analizează căile de acces. O aplicație poate depăși această limită de *MAX_PATH* caractere, trimițând nume de cale mai lungi prin apelarea versiunii lărgite (W) a funcției *GetFileAttributes* cu inserarea secvenței „\\?\" care comunică funcției să deconecteze analiza căii. Aceasta permite căilor mai mari decât *MAX_PATH* să fie utilizate cu *GetFileAttributesW*. Procedeul funcționează și cu numele de tip UNC (*Uniform Naming Convention* – sistem de numire a fișierelor într-o rețea de calculatoare). Sistemul de operare ignoră „\\?\" ca parte a căii de acces. De exemplu, „\\?\C:\mydocuments\private" este văzută ca „C:\mydocuments\private", iar „\\?\UNC\teora\redact\compendiu" este văzută ca „\\teora\redact\compendiu". Sub Windows 95, șirul *lpFileName* nu trebuie să depășească *MAX_PATH* caractere. Windows 95 nu acceptă prefixul „\\?\".

Dacă funcția reușește, valoarea returnată conține atributele fișierului sau directorului specificat. Dacă funcția eșuează, valoarea returnată va fi *0xFFFFFFFF*. Pentru informații suplimentare apelați *GetLastError*.

Atributele pot lua una sau mai multe din valorile enumerate mai jos:

Valoare	Descriere
FILE_ATTRIBUTE_ARCHIVE	Fișierul sau directorul este un fișier sau director arhivă. Aplicațiile utilizează acest indicator pentru marcarea fișierelor pentru *backup* sau eliminare.
FILE_ATTRIBUTE_COMPRESSED	Fișierul sau directorul sunt comprimate. Pentru un fișier aceasta înseamnă că toate datele din fișier sunt comprimate. Pentru un director, aceasta înseamnă că atributul comprimat este implicit pentru fișierele și directoarele nou create.
FILE_ATTRIBUTE_DIRECTORY	Obiectul curent este un director.

(continuare)

Valoare	Descriere
FILE_ATTRIBUTE_HIDDEN	Fişierul sau directorul este ascuns. Nu sunt incluse în enumerarea conţinutului directorului.
FILE_ATTRIBUTE_NORMAL	Fişierul sau directorul nu are alt set de atribute. Este valid doar dacă este singurul folosit.
FILE_ATTRIBUTE_OFFLINE	Datele din fişier nu sunt imediat disponibile. Precizează că datele fişierului au fost fizic mutate într-o stocare offline.
FILE_ATTRIBUTE_READONLY	Fişierul sau directorul este read-only. Aplicaţiile pot citi fişierul dar nu îl pot scrie sau şterge. În cazul unui director, aplicaţiile nu îl pot şterge.
FILE_ATTRIBUTE_SYSTEM	Fişierul sau directorul este parte a sistemului de operare sau este utilizat exclusiv de sistemul de operare.
FILE_ATTRIBUTE_TEMPORARY	Fişierul este utilizat pentru stocare temporară. Fişierul temporar trebuie şters de către aplicaţie imediat ce nu mai este nevoie de el.

Tabelul 1464 *Valorile returnate de funcţia* ***GetFileAttributes***.

Programele dumneavoastră pot utiliza funcţia *GetFileAttributes* pentru obţinerea atributelor unui fişier. Uneori însă, va trebui ca programele să modifice aceste atribute. În acest scop puteţi utiliza funcţia *SetFileAttributes* pentru stabilirea atributelor unui fişier. Veţi implementa funcţia *SetFileAttributes* potrivit prototipului prezentat mai jos:

```
BOOL SetFileAttributes(
    LPCTSTR lpFileName,      // adresa numelui fisierului
    DWORD dwFileAttributes   // atribute de stabilit
);
```

Parametrul *lpFileName* indică un şir care specifică numele fişierului ale cărui atribute urmează a fi stabilite. Aceleaşi limitări se aplică parametrului *lpFileName* ca şi parametrului *lpFileName* din funcţia *GetFileAttributes*. Parametrul *dwFileAttributes* specifică atributele care vor fi aplicate fişierului. Acest parametru poate fi o combinaţie a valorilor prezentate în Tabelul 1464 de mai sus. Însă, orice alt atribut suprascrie valoarea *FILE_ATTRIBUTE_NORMAL*.

Dacă funcţia reuşeşte, valoarea returnată va fi diferită de zero. Dacă funcţia eşuează, valoarea returnată va fi zero. Pentru mai multe informaţii despre erori, apelaţi funcţia *GetLastError*. Nu puteţi utiliza funcţia *SetFileAttribute* pentru stabilirea stării de comprimare a unui fişier. Fixarea valorii *FILE_ATTRIBUTE_COMPRESSED* în parametrul *dwFileAttributes* nu are nici un efect. Pentru stabilirea stării de comprimare a fişierului, utilizaţi funcţia *DeviceIoControl* şi operaţia *FSCTL_SET_COMPRESSION*.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Check_ReadOnly.cpp*. Programul verifică fişierul *file.dat* pentru a vedea dacă a fost stabilit atributul de protejare la scriere (read-only). Dacă da, programul elimină toate atributele fişierului şi apoi şterge fişierul.

1465 OBŢINEREA DIMENSIUNII UNUI FIŞIER



Windows ataşează anumite atribute fiecărui fişier pe care îl salvează pe disc. Una dintre cel mai obişnuite cereri efectuate de programe este obţinerea dimensiunii fişierului. Funcţia

GetFileSize regăsește dimensiunea, în octeți, a unui anumit fișier. Veți implementa funcția *GetFileSize* potrivit prototipului prezentat mai jos:

```
DWORD GetFileSize(
    HANDLE hFile,           // identificatorul de fisier
    LPDWORD lpFileSizeHigh  // adresa cuvantului mai
                            // semnificativ cu dimensiunea
                            // de fisier
);
```

Parametrul *hFile* specifică un identificator deschis pentru fișierul a cărui dimensiune va fi returnată. Identificatorul trebuie să fi fost creat cu drepturile de acces *GENERIC_READ* sau *GENERIC_WRITE*. Parametrul *lpFileSizeHigh* indică variabila unde va fi returnat cuvântul mai semnificativ al dimensiunii fișierului. Dacă aplicația nu necesită cuvântul mai semnificativ, parametrul poate fi NULL.

Dacă funcția reușește, valoarea returnată este cuvântul mai puțin semnificativ al dimensiunii fișierului, iar dacă *lpFileSizeHigh* nu este NULL, funcția depune cuvântul mai semnificativ al dimensiunii fișierului în variabila indicată de acest parametru. Dacă funcția eșuează, iar *lpFileSizeHigh* este NULL, valoarea returnată este *0xFFFFFFFF*. Pentru informații suplimentare apelați *GetLastError*. Dacă funcția eșuează și *lpFileSizeHigh* nu este NULL, valoarea returnată este *0xFFFFFFFF*, iar *GetLastError* va returna altă valoare decât *NO_ERROR*.

Nu puteți utiliza funcția *GetFileSize* cu un identificator de dispozitiv fără căutare cum ar fi un dispozitiv de comunicație sau un canal de transfer. Pentru obținerea tipului fișierului, utilizați funcția *GetFileType*. *GetFileSize* obține dimensiunea necomprimată a unui fișier. Utilizați *GetCompressedFileSize* pentru obținerea dimensiunii unui fișier comprimat. Rețineți că, dacă valoarea returnată este *0xFFFFFFFF*, iar *lpFileSizeHigh* nu este NULL, aplicația trebuie să apeleze *GetLastError* pentru a stabili dacă funcția a reușit sau a eșuat.

CD-ROM-ul care însoțește cartea de față conține programul *Check_FileSize.cpp*. Acest program definește funcția *DisplayFileSize* care mai întâi utilizează funcția *GetFileType* pentru a afla tipul identificatorului de fișier. Dacă fișierul este un fișier pe disc, funcția *DisplayFileSize* apelează *GetFileSize* pentru obținerea dimensiunii fișierului și afișează dimensiunea într-o casetă de mesaj. Dacă fișierul nu este un fișier pe disc, funcția avertizează utilizatorul în legătură cu acest fapt.

OBȚINEREA MARCAJULUI DE TIMP AL UNUI FIȘIER

C/C++1466

În câteva secțiuni anterioare ați învățat cum să aflați marcajul timpului fișierelor utilizând funcțiile C de run-time. Interfața Windows API, la rândul ei, pune la dispoziție o funcție care permite programelor să regăsească marcajul de timp al unui fișier. Funcția *GetFileTime* regăsește data și ora la care a fost creat un fișier, când a fost pentru ultima dată accesat sau pentru ultima dată modificat. Prototipul funcției *GetFileTime* este:

```
BOOL GetFileTime(
    HANDLE hFile,                 // identifica fisierul
    LPFILETIME lpCreationTime,    // adresa cu momentul crearii
```

```
    LPFILETIME lpLastAccessTime, // adresa cu momentul
                                 // ultimului acces
    LPFILETIME lpLastWriteTime   // adresa cu momentul ultimei
                                 // scrieri
);
```

Parametrul *hFile* identifică fişierele de la care se obţin data şi ora. Identificatorul de fişier trebuie să fi fost scris cu dreptul de acces *GENERIC_READ*. Parametrul *lpCreationTime* indică o structură *FILETIME* pentru regăsirea datei şi orei la care a fost creat fişierul. Acest parametru trebuie să fie NULL, dacă aplicaţia nu necesită aceste informaţii. Parametrul *lpLastAccessTime* indică o structură *FILETIME* pentru regăsirea datei şi orei la care fişierul a fost ultima oară accesat. Ultimul moment de acces cuprinde ultima dată şi oră la care fişierul a fost scris, citit sau, în cazul fişierelor executabile, rulat. Acest parametru poate fi NULL dacă aplicaţia nu necesită aceste informaţii. Parametrul *lpLastWriteTime* indică o structură *FILETIME* pentru regăsirea datei şi orei la care fişierul a fost ultima oară scris. Acest parametru poate fi NULL dacă aplicaţia nu necesită aceste informaţii.

Dacă funcţia reuşeşte, valoarea returnată va fi diferită de zero, iar dacă eşuează, va returna valoarea zero. Pentru informaţii suplimentare, apelaţi *GetLastError*.

Sistemele de fişiere din Windows 95 şi Windows NT acceptă marcajul de timp pentru crearea, ultima accesare şi ultima operaţie de scriere în fişiere. În Windows 95 precizia orei în sistemul de fişiere este de 2 secunde (adică momentul înregistrat de Windows în atribut va fi într-un interval de două secunde faţă de momentul real al accesului). Precizia timpului în alte sisteme de fişiere, cum sunt cele conectate în reţea, depinde de sistemul de fişiere de la distanţă. Precizia timpului poate fi limitată de dispozitivul aflat la distanţă.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Get_Time.cpp* care utilizează *GetFileTime* pentru obţinerea orei curente a fişierului şi convertirea ei la ora locală. Apoi programul converteşte ora locală a structurii *FILETIME* în formate de dată şi oră recunoscute de DOS. În final, programul afişează, într-o casetă de mesaj, data şi timpul din DOS, convertite.

1467 CREAREA DIRECTOARELOR



În secţiunea 396, aţi învăţat cum se utilizează o funcţie standard de bibliotecă run-time C pentru a crea un director în DOS. În mod asemănător, programele dumneavoastră pot utiliza funcţia API Win32 *CreateDirectory* pentru a crea un nou director. Dacă sistemul de fişiere de bază acceptă securitatea fişierelor şi directoarelor, funcţia aplică un descriptor de securitate specificat noului director. Amintiţi-vă, fiecare proces va avea propriul său director curent, astfel că apelările funcţiei *CreateDirectory* nu ar trebui să presupună că directorul curent al procesului este o constantă. Veţi utiliza funcţia *CreateDirectory* în cadrul programelor dumneavoastră, cum arătăm în următorul prototip:

```
BOOL CreateDirectory(
    LPCTSTR lpPathName, // pointer la sirul cale de acces
                        // director
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
// pointer la descriptor de securitate
);
```

Parametrul *lpPathName* indică un șir de caractere terminat în *NULL* care conține calea de acces a directorului care va fi creat. Există o limită implicită a dimensiunii șirului pentru căile de acces, de *MAX_PATH* caractere. Această limită este corelată cu modul în care funcția *CreateDirectory* analizează căile. Puteți depăși limitele căii de acces sub WindowsNT, așa cum este prezentat în secțiunea 1464.

Parametrul *lpSecurityAttributes* este un pointer către o structură *SECURITY_ATTRIBUTES* care determină dacă procesele copil pot moșteni identificatorul returnat. Dacă *lpSecurityAttributes* este *NULL*, procesele copil nu îl pot moșteni. Sub Windows NT, membrul *lpSecurityDescriptor* al structurii specifică un descriptor de securitate pentru noul director. Dacă *lpSecurityAttributes* este *NULL*, directorul obține un descriptor implicit de securitate. Sistemul de fișiere de destinație trebuie să accepte securitatea fișierelor și a directoarelor pentru ca acest parametru să aibă un efect. Sub Windows 95, sistemul de operare ignoră membrul *lpSecurityDescriptor* al structurii.

Dacă funcția reușește execuția, valoarea returnată este diferită de zero. Dacă funcția eșuează, valoarea returnată este zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*.

CD-ROM-ul care însoțește această carte cuprinde programul *Create_NewDir.cpp*. Programul utilizează funcția *CreateDirectory* pentru a crea un nou director numit „New Directory" în unitatea C: a calculatorului pe care programul rulează la acel moment.

Observație: *Anumite sisteme de fișiere, cum ar fi NTFS (NT File System) acceptă comprimarea individuală a fișierelor și directoarelor. Pe volume formatate pentru un astfel de sistem de fișiere, un nou director moștenește atributul de comprimare de la directorul părinte. O aplicație poate apela* ***CreateFile*** *cu indicatorul* ***FILE_FLAG_BACKUP_SEMANTICS*** *stabilit pentru a obține un identificator pentru un director. Pentru un exemplu de cod, vedeți* ***CreateFile.***

OBȚINEREA ȘI STABILIREA DIRECTORULUI CURENT

C/C++1468

După cum ați învățat în secțiunea 1467, programele dumneavoastră pot crea noi directoare în cadrul sistemului de fișiere. Însă, programele dumneavoastră vor putea, de asemenea, să ceară informații despre directorul curent mult mai des decât vor crea noi directoare. Funcția *GetCurrentDirectory* află directorul curent pentru procesul curent. Veți utiliza funcția *GetCurrentDirectory* în cadrul programelor dumneavoastră așa cum arătăm în următorul prototip:

```
DWORD GetCurrentDirectory(
    DWORD nBufferLength,   // dimensiune buffer director, in
                           // caractere
    LPTSTR lpBuffer        // adresa buffer pt. directorul curent
);
```

Parametrul *nBufferLength* specifică lungimea, în caractere, a bufferului pentru șirul de caractere al directorului curent. Lungimea bufferului trebuie să cuprindă spațiu pentru caracterul de încheiere *NULL*. Parametrul *lpBuffer* este un pointer către bufferul șirului de caractere al directorului curent. Acest șir terminat în *NULL* conține calea de acces absolută pentru directorul curent. Dacă funcția reușește, valoarea returnată specifică numărul de

caractere scrise în buffer, exclusiv caracterul de încheiere *NULL*. Dacă funcția eșuează, valoarea returnată este zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*. Dacă bufferul indicat de *lpBuffer* nu are o dimensiune suficientă, valoarea returnată specifică dimensiunea necesară a bufferului, inclusiv numărul de octeți necesari pentru caracterul de încheiere NULL.

În mod asemănător, programele dumneavoastră pot utiliza funcția *SetCurrentDirectory* pentru a schimba directorul curent al procesului curent, cu altul specificat de dumneavoastră. Veți utiliza funcția *SetCurrentDirectory* în cadrul programelor dumneavoastră așa cum arătăm în următorul prototip:

```
BOOL SetCurrentDiretory(LPCTSTR lpPathName);
```

Parametrul *lpPathName* indică un șir de caractere terminat în *NULL* care conține calea pentru noul director curent. Acest parametru poate fi o cale relativă sau o cale complet definită. În fiecare dintre aceste cazuri, sistemul de operare calculează calea complet definită a directorului specificat și păstrează această cale ca director curent. Dacă funcția reușește, ea returnează o valoare diferită de zero. Dacă funcția eșuează, ea returnează valoarea zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*.

Fiecare proces are un singur director curent, alcătuit din două părți:

- Un nume de disc, care este fie litera unității urmată de două puncte, fie numele serverului și numele de partajare (de exemplu, \\servername\sharename)
- Un director din acel disc

CD-ROM-ul care însoțește această carte cuprinde programul *Create_Set.cpp*. Atunci când compilați și executați programul, acesta va afișa directorul curent, apoi va crea directorul *New Directory* și, în sfârșit, va schimba directorul curent cu *New Directory* și va afișa noul director curent.

1469 OBȚINEREA DIRECTOARELOR WINDOWS ȘI DE SISTEM



În secțiunea 1468, ați învățat cum pot programele dumneavoastră să obțină directorul curent al procesului curent. Pe măsură ce programele dumneavoastră devin mai complexe, veți solicita deseori informații despre locația directorului *Windows* și locația pentru directorul *system* din Windows. Funcția *GetWindowsDirectory* află calea directorului Windows (care poate fi un director cu un alt nume decât *c:\windows* sau *c:\winnt*). Directorul Windows conține fișiere precum aplicațiile Windows, fișiere de inițializare și fișiere Help. Veți utiliza funcția *GetWindowsDirectory* în cadrul programelor dumneavoastră așa cum arătăm în următorul prototip:

```
UINT GetWindowsDirectory(
    LPTSTR lpBuffer, // adresa de buffer a directorului Windows
    UINT uSize       // dimensiune buffer director
);
```

Parametrul *lpBuffer* indică bufferul pentru a primi șirul de caractere terminat în *NULL* care conține calea. Această cale nu trebuie să se încheie cu backslash, cu excepția cazului în care directorul Windows este director rădăcină. De exemplu, dacă directorul Windows este

denumit windows pe unitatea C, calea directorului Windows găsită de această funcție este c:\windows. Dacă Windows a fost instalat în directorul rădăcină al unității C, calea găsită este C:\. Parametrul uSize specifică dimensiunea maximă, în caractere, a bufferului specificat de către parametrul *lpBuffer*. Ar trebui să stabiliți parametrul *uSize* la cel puțin valoarea *MAX_PATH* pentru a aloca spațiu suficient în buffer pentru cale.

Dacă funcția reușește, valoarea returnată este lungimea, în caractere, a șirului pe care funcția l-a copiat în buffer, exclusiv caracterul de încheiere *NULL*. Dacă lungimea este mai mare decât dimensiunea bufferului, funcția returnează dimensiunea bufferului cerută pentru a păstra calea. Dacă funcția eșuează, funcția returnează zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*.

Directorul Windows este directorul în care o aplicație ar trebui să stocheze fișierele de inițializare și fișierele Help. Dacă utilizatorul rulează o versiune partajată de Windows, sistemul de operare garantează că directorul windows este privat pentru fiecare utilizator. Dacă o aplicație creează alte fișiere pe care doriți să le stocați numai pentru uzul utilizatorului, ar trebui să le plasați în directorul specificat de variabila de mediu *HOMEPATH*. *HOMEPATH* specifică întotdeauna fie directorul gazdă (home) al utilizatorului, care este garantat ca privat pentru fiecare utilizator, fie directorul implicit (de exemplu, c:\users\default) unde va avea acces fiecare utilizator.

La fel cum programele dumneavoastră cer deseori informații despre locația directorului Windows, ele vor cere și informații despre directorul de sistem. Funcția *GetSystemDirectory* găsește calea directorului de sistem din Windows. Directorul de sistem conține fișiere ca biblioteci Windows, drivere, fișiere de fonturi. Veți utiliza funcția *GetSystemDirectory* în cadrul programelor dumneavoastră așa cum arătăm în următorul prototip:

```
UINT GetSystemDirectory(
    LPTSTR lpBuffer, // adresa bufferului pt. directorul sistem
    UINT uSize       // dimensiune buffer director
);
```

Parametrii funcției *GetSystemDirectory* sunt aceiași cu cei ai funcției *GetWindowsDirectory*. Dacă funcția *GetSystemDirectory* reușește, ea returnează lungimea, în caractere, a șirului de caractere copiat în buffer, exclusiv caracterul *NULL*. Dacă lungimea este mai mare decât dimensiunea bufferului, funcția returnează dimensiunea bufferului necesară pentru a păstra calea. Dacă funcția eșuează, ea returnează zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*.

CD-ROM-ul care însoțește această carte cuprinde programul *Show_Windows.cpp*. Atunci când compilați și executați programul *Show_Windows.cpp*, acesta va utiliza funcțiile *GetWindowsDirectory* și *GetSystemDirectory* pentru a obține numele directoarelor Windows și system. Programul va afișa apoi numele directoarelor în cadrul zonei client a ferestrei.

Observație: *De regulă, aplicațiile ar trebui să nu creeze fișiere în directorul* ***system***. *Dacă utilizatorul rulează o versiune partajată de Windows, aplicația nu trebuie să aibă acces pentru scriere la directorul* ***system***. *Aplicațiile trebuie să creeze fișiere numai în directorul pe care îl returnează funcția* ***GetWindowsDirectory***.

1470 ȘTERGEREA DIRECTOARELOR



ι fel cum programele dumneavoastră pot crea directoare temporare sau directoare pe care le utilizează numai intern, va fi probabil necesar uneori ca programele dumneavoastră să eargă un director existent. Funcția *RemoveDirectory* șterge un director existent, golit. Veți iliza funcția *RemoveDirectory* așa cum arătăm în următorul prototip:

```
BOOL RemoveDirectory( LPCTSTR lpPathName );
```

ırametrul *lpPathName* indică un șir de caractere terminat în *NULL* care conține calea irectorului ce urmează a fi șters. Calea trebuie să specifice un director golit, iar procesul de ›elare trebuie să aibă acces de ștergere la director. Dacă funcția reușește, ea returnează o ıloare diferită de zero. Dacă funcția eșuează, ea va returna valoarea zero. Pentru a obține ıai multe informații despre eroare, apelați *GetLastError*.

D-ROM-ul care însoțește această carte cuprinde programul *Create_Remove.cpp*. Atunci ind compilați și executați programul *Create_Remove.cpp*, el va crea un nou director ːnumit *Child Folder* și va atenționa utilizatorul că a făcut aceasta. Apoi, programul va șterge irectorul recent creat și va atenționa, de asemenea, utilizatorul despre modificarea ıtervenită.

1471 COPIEREA FIȘIERELOR



rogramele dumneavoastră pot manipula fișiere pe scară largă în Windows. Pe măsură ce rogramele dumneavoastră devin tot mai complexe, va trebui uneori, probabil, să copiați un șier dintr-o locație într-o alta, menținând fișierul inițial în locația sa inițială. Funcția *CopyFile* opiază un fișier existent într-un nou fișier. Veți utiliza funcția *CopyFile* așa cum arătăm în rototipul următor:

```
BOOL CopyFile(
    LPCTSTR lpExistingFileName,   // pointer la numele
                                  // fisierului existent
    LPCTSTR lpNewFileName,        // pointer la numele
                                  // fisierului pt. copiere
    BOOL bFailIfExists            // indicator pentru operatii
                                  // daca fisierul exista
);
```

arametrul *lpExistingFileName* indică șirul de caractere terminat în *NULL* care specifică numele ıui fișier existent. Parametrul *lpNewFileName* indică șirul de caractere terminat în *NULL* care ›ecifică numele noului fișier. Parametrul *bFailIfExists* specifică modul în care funcția va opera ıcă există deja un fișier cu același nume cu cel specificat de *lpNewFileName*. Dacă parametrul *FailIfExists* este *True* și noul fișier deja există, funcția eșuează. Dacă acest parametru este *alse* și noul fișier deja există, funcția suprascrie fișierul existent și se încheie cu succes.

ıacă funcția reușește, ea returnează o valoare diferită de zero. Dacă funcția eșuează, ea ːturnează zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*. tributele de securitate pentru fișierul existent nu sunt copiate în noul fișier.

tributele fișierului (*FILE_ATTRIBUTE_**) pentru fișierul existent sunt copiate în noul fișier. ›e exemplu, dacă fișierul existent are atributul de fișier *FILE_ATTRIBUTE_READONLY*, copia

creată prin apelarea lui *CopyFile* va avea, de asemenea, atributul de fișier *FILE_ATTRIBUTE_READONLY*.

CD-ROM-ul care însoțește această carte cuprinde programul *Create_Copy.cpp*. Atunci când compilați și executați programul *Create_Copy.cpp*, acesta va crea fișierul *file1.txt*. atunci când utilizatorul selectează opțiunea *Test!*, programul va copia fișierul *file1.txt* în fișierul *file2.txt*. Dacă fișierul *file2.txt* există deja, programul va întreba utilizatorul dacă să înlocuiască fișierul existent.

MUTAREA ȘI REDENUMIREA FIȘIERELOR

C/C++1472

După cum ați învățat în secțiunea 1471, programele dumneavoastră pot să facă ușor o copie a unui fișier și să o plaseze într-o altă locație. Însă, programele dumneavoastră pot să mute un fișier sau un director într-o altă locație, fără să păstreze copia inițială a fișierului. Funcția *MoveFile* redenumește un fișier sau un director existent (inclusiv toți copiii săi). Veți utiliza funcția *MoveFile* în cadrul programelor dumneavoastră așa cum arătăm în prototipul următor:

```
BOOL MoveFile(
   LPCTSTR lpExistingFileName, // adresa cu numele fisierului
                               // existent
    LPCTSTR lpNewFileName       // adresa cu noul nume al
                               // fisierului
);
```

Parametrul *lpExistingFileName* indică un șir de caractere terminat în *NULL* care numește un fișier sau director existent. Parametrul *lpNewFileName* indică un șir de caractere terminat în *NULL* care specifică noul nume al fișierului sau directorului. Noul nume nu trebuie să existe deja. Noul fișier poate fi pe un alt sistem de fișiere sau unitate. Un nou director trebuie să se afle pe aceeași unitate. Dacă funcția reușește, ea va returna o valoare diferită de zero. Dacă funcția eșuează, ea va returna valoarea zero. Pentru a obține mai multe informații despre eroare, apelați *GetLastError*.

Funcția *MoveFile* va muta (redenumi) fie un fișier, fie un director (inclusiv toți descendenții săi), fie în cadrul aceluiași director, fie în afara lui. Unica limitare a funcției *MoveFile* este aceea că va eșua dacă destinația directorului este pe alt volum, atunci când se mută un director.

CD-ROM-ul care însoțește această carte cuprinde programul *Move_Folder.cpp*. Atunci când compilați și executați programul *Move_Folder.cpp*, acesta va crea mai întâi o structură temporară de director. Atunci când utilizatorul selectează opțiunea *Test!*, programul va muta unul dintre directoare din structura de director în alt director.

ȘTERGEREA FIȘIERELOR

C/C++1473

În secțiunea 1470, ați învățat cum pot utiliza programele dumneavoastră funcția *RemoveDirectory* pentru a șterge un director dintr-un sistem de fișiere. Însă, așa cum se remarcă în secțiunea 1470, directorul trebuie să fie golit înainte de a invoca funcția *RemoveDirectory*, altfel funcția va eșua. Pentru a elimina fișiere dintr-un director, programele dumneavoastră pot utiliza funcția *DeleteFile*. Veți utiliza funcția *DeleteFile* în cadrul programelor dumneavoastră așa cum arătăm în următorul prototip:

```
BOOL DeleteFile( LPCTSTR lpFileName );
```

Parametrul *lpFileName* indică un şir de caractere terminat în *NULL* care specifică fişierul pe care funcţia îl va şterge. Dacă funcţia reuşeşte, ea va returna o valoare diferită de zero. Dacă funcţia eşuează, ea va returna valoarea zero. Pentru a obţine mai multe informaţii despre eroare, apelaţi *GetLastError*.

Dacă o aplicaţie încearcă să şteargă un fişier care nu există, funcţia *DeleteFile* va eşua. Sub Windows 95, funcţia *DeleteFile* şterge un fişier chiar dacă fişierul este deschis la acel moment pentru operaţii obişnuite de intrare şi ieşire sau ca fişier mapat în memorie. Pentru a preveni erorile, închideţi fişierele înainte de a încerca să le ştergeţi. Sub Windows NT, funcţia *DeleteFile* eşuează dacă o aplicaţie încearcă să şteargă un fişier care este deschis la acel moment pentru operaţii obişnuite de I/O sau ca fişier mapat în memorie.

1474 UTILIZAREA FUNCŢIEI FINDFIRSTFILE PENTRU LOCALIZAREA FIŞIERELOR

C/C++

În secţiunile 390 şi 381, aţi învăţat cum să utilizaţi funcţiile din biblioteca *run-time* standard de C pentru căutarea căii de comandă şi a directorului curent al unui fişier. În cadrul programelor Windows veţi utiliza funcţiile *Find* pentru localizarea fişierelor care corespund unor criterii date. Programele dumneavoastră pot utiliza două funcţii *Find*. Funcţia *FindFirstFile* caută într-un director fişierul care corespunde unui nume de fişier specificat. *FindFirstFile* examinează numele de subdirectoare şi numele de fişiere. Veţi implementa funcţia *FindFirstFile* potrivit prototipului prezentat mai jos:

```
HANDLE FindFirstFile(
    LPCTSTR lpFileName, // pointer la numele fisierului cautat
    LPWIN32_FIND_DATA lpFindFileData   // pointer la informatia
                                       // returnata
);
```

Sub Windows 95 şi Windows NT, parametrul *lpFileName* indică un şir terminat cu NULL care specifică un nume valid de director sau cale şi numele fişierului, care poate conţine caractere de înlocuire (* şi ?). Sub Windows 95, acest şir nu trebuie să depăşească un număr de *MAX_PATH* caractere.

Parametrul *lpFindFileData* indică o structură *WIN32_FIND_DATA* care primeşte informaţia despre fişierul sau directorul găsit. Structura poate fi folosită în apeluri ulterioare la funcţiile *FindNextFile* sau *FindClose*, pentru accesul la fişier sau subdirector. Structura *WIN32_FIND_DATA* descrie fişierul găsit cu funcţiile *FindFirstFile, FindFirstFileEx sau FindNextFile*. Interfaţa *Win32* defineşte structura *WIN32_FIND_DATA* astfel:

```
typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
```

```
    DWORD dwReserved0;
    DWORD dwReserved1;
    TCHAR cFileName[ MAX_PATH ];
    TCHAR cAlternateFileName[ 14 ];
} WIN32_FIND_DATA;
```

Tabelul 1474 explică în detaliu structura *WIN32_FIND_DATA*.

Membru	Descriere
dwFileAttributes	Specifică atributele fişierului găsit. Acest membru poate lua una sau mai multe din valorile prezentate în Tabelul 1464.
ftCreationTime	Specifică o structură *FILETIME* care conţine ora când a fost creat fişierul. *FindFirstFile* şi *FindNextFile* raportează momentele în format UTC (*Coordinated Universal Time*). Aceste funcţii fixează membrii structurii *FILETIME* la zero dacă sistemul de fişiere ce conţine fişierul nu acceptă acest membru de timp. Puteţi utiliza *FileTimeToLocalFileTime* pentru convertirea din UTC la ora locală şi apoi funcţia *FileTimeToSystemTime* pentru convertirea timpului local la o structură *SYSTEMTIME* care conţine membri individuali pentru lună, zi, an, zi din săptămână, oră, minut, secundă şi milisecundă.
ftLastAccessTime	Specifică o structură *FILETIME* care conţine ora când a fost ultima dată accesat fişierul. Momentul este prezentat format UTC; membrii structurii *FILETIME* sunt zero dacă sistemul de fişiere nu acceptă acest membru de timp.
ftLastWriteTime	Specifică o structură *FILETIME* care conţine ora la care s-a scris ultima dată în fişier. Momentul este prezentat format UTC; membrii structurii *FILETIME* sunt zero dacă sistemul de fişiere nu acceptă acest membru de timp.
nFileSizeHigh	Conţine cuvântul mai semnificativ al valorii *DWORD* a dimensiunii fişierului, în octeţi. Valoarea este zero dacă dimensiunea fişierului nu este mai mare decât *MAXDWORD*. Dimensiunea fişierului este egală cu *(nFileSizeHigh * MAXDWORD) + nFileSizeLow*.
nFileSizeLow	Conţine cuvântul mai puţin semnificativ al valorii *DWORD* a dimensiunii fişierului, în octeţi.
dwReserved0	Rezervat pentru versiuni viitoare.
dwReserved1	Rezervat pentru versiuni viitoare.
cFileName	Un şir terminat cu NULL care reprezintă numele fişierului.
cAlternateFileName	Un şir terminat cu NULL care este o alternativă a numelui de fişier. Acest nume este în formatul clasic de nume 8.3 (opt caractere pentru nume, 3 caractere pentru extensie).

Tabelul 1474 *Componentele structurii* ***WIN32_FIND_DATA***.

Dacă un fişier are un nume lung, numele complet apare în câmpul *cFileName*, iar versiunea trunchiată de 8.3 apare în câmpul *cAlternateFileName*. Altfel, *cAlternateFileName* este gol. Ca alternativă, puteţi utiliza funcţia *GetShortPathName* pentru obţinerea versiunii de format 8.3 a numelui de fişier.

Dacă funcţia reuşeşte, valoarea returnată este identificatorul de căutare utilizat în apelul ulterior la funcţiile *FindNextFile* sau *FindClose*. Dacă funcţia eşuează, valoarea returnată este *INVALID_HANDLE_VALUE*. Pentru informaţii suplimentare apelaţi funcţia *GetLastError*.

Funcţia *FindFirstFile* deschide un identificator de căutare şi returnează informaţii despre primul fişier al cărui nume corespunde unui model specificat. O dată stabilit identificatorul de căutare, puteţi utiliza funcţia *FindNextFile* pentru căutarea altor fişiere care corespund aceluiaşi model. Când identificatorul de căutare nu mai e necesar, utilizaţi funcţia *FindClose* pentru închiderea lui. Această funcţie caută fişierele luând ca bază numai numele. Nu poate fi utilizată pentru căutările bazate pe atribute. În secţiunea 1477, găsiţi mai multe informaţii despre căutările bazate pe atribute.

1475 UTILIZAREA FUNCŢIEI FINDNEXTFILE



n secţiunea 1474, aţi învăţat despre funcţia *FindFirstFile* care caută într-un arbore director prima instanţă a unui fişier corespunzător unui nume de fişier dat. Însă, pentru continuarea căutării de alte fişiere cu acelaşi nume (sau care corespund caracterelor de înlocuire) trebuie apelată cea de-a doua funcţie *Find*. Funcţia *FindNextFile* continuă o căutare de fişier în urma unui apel anterior la funcţia *FindFirstFile*. Veţi implementa funcţia *FindNextFile* potrivit prototipului prezentat mai jos:

```
BOOL FindNextFile(
    HANDLE hFindFile,                    // identificator de cautare
    LPWIN32_FIND_DATA lpFindFileData // pointer la structura
                                     // pentru fisierul gasit
);
```

Parametrul *hFindFile* conţine un identificator de căutare returnat de un apel anterior la funcţia *FindFirstFile*. Parametrul *lpFindFileData* indică o structură *WIN32_FIND_DATA* care primeşte informaţii despre fişierul sau subdirectorul găsit. Structura poate fi utilizată în apeluri ulterioare la *FindNextFile* pentru referiri la fişierul sau subdirectorul găsit.

Dacă funcţia reuşeşte, valoarea returnată va fi diferită de zero. Dacă funcţia eşuează, valoarea returnată va fi zero. Pentru informaţii suplimentare apelaţi *GetLastError*. Dacă nu sunt găsite fişiere, funcţia *GetLastError* returnează *ERROR_NO_MORE_FILES*. Funcţia *FindNextFile* caută fişierele luând ca bază numai numele. Nu poate fi utilizată pentru căutările bazate pe atribute. Veţi afla mai multe despre atribute în secţiunea 1477.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Walk_Directories.cpp* care utilizează funcţiile *FindFirstFile* şi *FindNextFile* pentru căutări pe unitatea de disc pe care o specificaţi. În plus, programul utilizează un apel la o funcţie recursivă pentru a se asigura că se caută în toate directoarele unităţii curente. Pe măsură ce programul caută, el adaugă numele fişierelor în caseta listă afişată.

1476 ÎNCHIDEREA IDENTIFICATORULUI DE CĂUTARE CU FINDCLOSE



Când programele lucrează cu funcţiile *FindFirstFile* şi *FindNextFile*, veţi deschide un nou identificator (un identificator de căutare) pentru acele funcţii pe care sistemul de operare le returnează în mod normal la invocarea funcţiei *CreateFile*. Dacă încercaţi să închideţi un

identificator de căutare cu *CloseHandle*, programele dumneavoastră vor returna o eroare. În schimb, programele trebuie să utilizeze funcţia *FindClose* pentru închiderea identificatorului de căutare. Veţi implementa funcţia *FindClose* potrivit prototipului prezentat mai jos:

```
BOOL FindClose( HANDLE hFindFile );
```

Parametrul *hFindFile* conţine identificatorul de căutare. Acest identificator trebuie să fi fost deschis anterior de către funcţia *FindFirstFile*. Dacă funcţia reuşeşte, valoarea returnată va fi diferită de zero. Dacă funcţia eşuează, valoarea returnată va fi zero. După apelarea funcţiei *FindClose*

Identificatorul specificat de parametrul *hFindFile* nu poate fi utilizat în apeluri ulterioare la *FindNextFile* sau *FindClose*. În programul *Walk_Directories.cpp* descris în secţiunea 1475, programul utilizează funcţia *FindClose* numai după ce utilizatorul a încheiat parcurgerea tuturor directoarelor.

CĂUTAREA ATRIBUTELOR CU FUNCŢIILE FINDFILE

C/C++ 1477

Aşa cum aţi învăţat în secţiunea 1475, programele dumneavoastră pot utiliza o funcţie recursivă (*WalkDirRecurse* din programul *Walk_Directories.cpp*) pentru căutarea unui fişier în întreaga unitate de disc. Dacă, în schimb, veţi să proiectaţi o aplicaţie numai pentru Windows NT 4.0, puteţi utiliza funcţia *FindFirstFileEx* (împreună cu funcţia *FindNextFile*) pentru căutarea într-un director. Funcţia caută într-un director un fişier ale cărui nume şi atribute corespund celor specificate în apelul funcţiei. Veţi implementa funcţia *FindFirstFileEx* potrivit prototipului prezentat mai jos:

```
HANDLE FindFirstFileEx(
    LPCTSTR lpFileName, // pointer la numele fisierului de cautat
    FINDEX_INFO_LEVELS fInfoLevelId, // nivel de informatie al
                                     // datelor returnate
    LPVOID lpFindFileData, // pointer la informatia returnata
    FINDEX_SEARCH_OPS fSearchOp, // tip de filtrare
    LPVOID lpSearchFilter, // pointer la criteriile de cautare
    DWORD dwAdditionalFlags // indicatoare de control pentru
                            // cautare suplimentara
);
```

Parametrul *lpFileName* indică un şir terminat cu NULL care specifică un director sau cale de acces validă şi numele fişierului, care pot conţine caractere de înlocuire (* şi ?). Parametrul *fInfoLevelId* specifică un tip de enumerare *FINDEX_INFO_LEVELS* care oferă nivelul de informaţie al datelor returnate. Parametrul *lpFindFileData* returnează un pointer la datele din fişier. Tipul pointerului este determinat de nivelul de informaţie specificat în parametrul *fInfoLevelId*. Parametrul *fSearchOp* specifică un tip de enumerare *FINDEX_SEARCH_OPS* care oferă tipul de filtrare ce trebuie efectuat după compararea caracterelor de înlocuire. Parametrul *lpSearchFilter* arată criteriile de căutare, dacă parametrul *fSearchOp* de tip enumerare specifică necesitatea unor informaţii de căutare structurate. Momentan, nici una din valorile acceptate de *fSearchOp* nu cere informaţii de căutare extinse (deşi versiunile viitoare de Windows NT vor necesita informaţii de căutare extinse). În consecinţă, pointerul trebuie să fie NULL. Parametrul *dwAdditionalFlags* specifică indicatoare suplimentare pentru

controlarea căutării. Puteți utiliza indicatorul *FIND_FIRST_EX_CASE_SENSITIVE* pentru căutări sensibile la diferențierile dintre majuscule și minuscule. Căutarea implicită nu este sensibilă la această diferențiere. Windows NT 4.0 nu definește nici un alt indicator, dar în versiunile viitoare vor fi.

Dacă funcția reușește, valoarea returnată este identificatorul de căutare care poate fi utilizat în apeluri ulterioare la funcțiile *FindNextFile* și *FindClose*. Dacă funcția eșuează, valoarea returnată este *INVALID_HANDLE_VALUE*. Funcția *FindFirstFileEx* vă permite deschiderea unui identificator de căutare și returnarea informațiilor despre primul fișier al cărui nume corespunde modelului și atributelor specificate. Dacă sistemul de fișiere de bază nu acceptă tipul specificat de filtrare în *fSearchOp*, altul decât filtrarea directorului, *FindFirstFileEx* eșuează cu eroarea *ERROR_NOT_SUPPORTED*. Aplicația trebuie, în acest caz, să utilizeze *FileExSearchNameMatch* și să efectueze propria ei filtrare. Consultați documentația online a compilatorului pentru mai multe informații despre *FileExSearchNameMatch*.

Când stabiliți identificatorul de căutare, aplicația îl poate utiliza cu funcția *FindNextFile* pentru căutarea altor fișiere care corespund aceluiași model ca în cazul funcțiilor care execută același tip de filtrare. Când identificatorul de căutare nu mai este necesar, el trebuie închis cu funcția *FindClose*.

Observație: *Pachetul SDK al produsului Windows NT 4.0 explică pe larg tipul enumerare pentru funcția* ***FindFirstFileEx***.

1478 UTILIZAREA FUNCȚIEI SEARCHPATH ÎN LOCUL FUNCȚIEI FIND PENTRU CĂUTĂRI

C/C++

Programele dumneavoastră pot utiliza funcțiile *Find* pentru căutarea unei serii de fișiere într-un director. Sau, programele pot utiliza funcția *SearchPath* pentru căutarea unui fișier specificat. Însă, *SearchPath* va căuta fișierul doar în cadrul unui anumit set de căi, așa cum se detaliază în Tabelul 1478. Veți implementa funcția *SearchPath* potrivit prototipului prezentat mai jos:

```
DWORD SearchPath(
    LPCTSTR lpPath,          // adresa caii de cautare
    LPCTSTR lpFileName,      // adresa numelui de fisier
    LPCTSTR lpExtension,     // adresa extensie
    DWORD nBufferLength,     // dimensiunea bufferului, in
                             // caractere
    LPTSTR lpBuffer,         // adresa de buffer pt.fisier gasit
    LPTSTR *lpFilePart       // adresa pointerului la componenta
                             // de fisier
);
```

Funcția *SearchPath* acceptă parametrii prezentați în Tabelul 1478.

Parametru	Descriere
lpPath	Indică un șir terminat cu NULL care conține calea pentru căutarea fișierului. Dacă acest parametru este NULL, funcția caută în următoarele directoare, în succesiunea descrisă mai jos: 1. Directorul din care a fost încărcată aplicația. 2. Directorul curent.

Parametru	Descriere
	3. Sub Windows 95, directorul sistem din Windows. Utilizați funcția *GetSystemDirectory* pentru obținerea căii acestui director. Sub Windows NT, directorul de sistem Windows pe 32 de biți. Utilizați funcția *GetSystemDirectory* pentru obținerea căii acestui director, denumit de regulă *SYSTEM32*.
	4. Sub Windows NT, directorul sistem pe 16 biți. Nu există nici o funcție Win32 care să obțină calea acestui director, dar oricum funcția *SearchPath* efectuează căutarea acestui director denumit, de regulă, *SYSTEM*.
	5. Directorul Windows. Utilizați funcția *GetWindowsDirectory* pentru obținerea căii acestui director.
	6. Directoarele enumerate în variabila de mediu *PATH*.
lpFileName	Indică un șir terminat cu NULL care specifică numele fișierului care urmează a fi căutat.
lpExtension	Indică un șir terminat cu NULL care specifică extensia adăugată fișierului în timpul căutării. Primul caracter al extensiei numelui de fișier trebuie să fie un punct (.). Extensia este adăugată numai dacă fișierul specificat nu se termină cu o extensie. Dacă extensia nu este necesară sau dacă numele de fișier conține deja o extensie, acest parametru este NULL.
nBufferLength	Specifică lungimea, în caractere, a bufferului care primește calea și numele de fișier valide.
lpBuffer	Indică bufferul cu calea și numele de fișier valide ale fișierului găsit.
lpFilePart	Indică adresa (în *lpBuffer*) ultimei componente cu calea și numele de fișier valide, care este adresa caracterului imediat următor caracterului backslash (\) final din cale.

***Tabelul 1478** Parametrii funcției **SearchPath**.*

Dacă funcția reușește, valoarea returnată este lungimea, în caractere, a șirului copiat în buffer, fără includerea terminatorului de șir NULL. Dacă valoarea returnată (adică lungimea șirului) este mai mare decât *nBufferLength*, valoarea returnată este dimensiunea bufferului cerut pentru păstrarea căii. Dacă funcția eșuează, valoarea returnată este zero. Pentru informații suplimentare despre eroare, apelați funcția *GetLastError*.

CD-ROM-ul care însoțește cartea de față conține programul *Search_For_Calc.cpp*. După compilare și lansare, programul *Search_For_Calc.cpp* va utiliza funcția *SearchPath* pentru căutarea fișierului *calc.exe* în unitatea respectivă. Dacă programul găsește fișierul *calc.exe*, el va afișa calea fișierului într-o casetă de mesaj.

OBȚINEREA UNEI CĂI DE ACCES TEMPORARE

C/C++1479

În secțiunile 386 și 387 ați învățat cum să creați un fișier temporar în programele C. Așa cum ați învățat, unul din pașii necesari creării unui fișier temporar este determinarea căii temporare din variabila de mediu *TEMP* sau *TMP*. În Windows, puteți utiliza funcția

GetTempPath pentru regăsirea căii directorului desemnat pentru fişierele temporare. Veţi implementa funcţia *GetTempPath* potrivit prototipului prezentat mai jos:

```
DWORD GetTempPath(
    DWORD nBufferLength, // dimensiunea bufferului, in
                         // caractere
    LPTSTR lpBuffer      // adresa de buffer pt. calea temporara
);
```

Parametrul *nBufferLength* specifică dimensiunea, în caractere, a bufferului cu şirul identificat de *lpBuffer*. Parametrul *lpBuffer* indică un buffer şir care primeşte şirul terminat cu NULL care conţine calea fişierului temporar.

Dacă funcţia *GetTempPath* reuşeşte, valoarea returnată este lungimea, în caractere, a şirului copiat în *lpBuffer*, fără includerea terminatorului de şir NULL. Dacă valoarea returnată (şirul căii) este mai mare decât *nBufferLength*, valoarea returnată este dimensiunea bufferului cerut pentru păstrarea căii. Dacă funcţia eşuează, valoarea returnată este zero.

Funcţia *GetTempPath* obţine calea fişierului temporar în modul prezentat mai jos:

1. Calea specificată de variabila de mediu TMP.
2. Calea specificată de variabila de mediu TEMP, dacă TMP nu este definit de Windows.
3. Directorul curent, dacă atât TMP cât şi TEMP nu sunt definite de Windows.

1480 CREAREA FIŞIERELOR TEMPORARE



În secţiunile 386 şi 387 aţi învăţat să creaţi fişiere temporare sub DOS. Crearea unui fişier temporar într-un program Windows este la fel de uşoară. Funcţia *GetTempFileName* creează un nume printr-un fişier temporar. Numele de fişier este concatenarea şirurilor căii şi prefixului specificate, un şir hexazecimal format dintr-un întreg specificat şi extensia .TMP. Întregul poate specifica o valoare diferită de zero, caz în care funcţia creează numele de fişier, dar nu creează fişierul. Dacă specificaţi zero ca valoare pentru întreg, funcţia creează un nume unic de fişier, realizând fişierul în directorul specificat. Veţi implementa funcţia *GetTempFileName* potrivit prototipului prezentat mai jos:

```
UINT GetTempFileName(
    LPCTSTR lpPathName,     // adresa numelui de director pt.
                            // fiser temporar
    LPCTSTR lpPrefixString, // adresa prefixului de nume de
                            // fisier
    UINT uUnique,           // numar pt. numele fisierului
                            // temporar
    LPTSTR lpTempFileName   // adresa de buffer care primeste
                            // noul nume de fisier
);
```

Parametrul *lpPathName* indică un şir terminat cu NULL, care conţine calea directorului pentru numele fişierului. Şirul trebuie să constea în caractere din setul ANSI. Aplicaţiile, de regulă, menţionează un punct (.) sau rezultatul funcţiei *GetTempPath* pentru acest parametru. Dacă *lpPathName* este NULL, funcţia eşuează. Parametrul *lpPrefixString* indică un şir

cu un prefix terminat cu NULL. Funcţia utilizează primele trei caractere ale şirului ca prefix pentru numele fişierului. Şirul trebuie să constea din caractere din setul ANSI. Parametrul *uUnique* specifică un întreg fără semn pe care funcţia îl converteşte la un şir hexazecimal pentru a fi utilizat la crearea numelui de fişier temporar. Dacă *uUnique* este diferit de zero, funcţia ataşează şirul hexazecimal la *lpPrefixString* pentru formarea numelui de fişier temporar. În acest caz, funcţia nu creează fişierul specificat şi nu testează dacă numele lui este unic. Dacă *uUnique* este zero, funcţia utilizează şirul hexazecimal derivat din ora curentă a sistemului. În acest caz, funcţia utilizează valori diferite până la găsirea unui nume de fişier unic şi apoi creează fişierul în directorul *lpPathName*. Parametrul *lpTempFileName* indică bufferul care primeşte numele fişierului temporar. Acest buffer este un şir terminat cu NULL şi constă în caractere din setul ANSI. Acest buffer trebuie aibă o lungime, în octeţi, de cel puţin *MAX_PATH* (de obicei 255) pentru acomodarea cu lungimea căii.

Dacă funcţia reuşeşte, valoarea returnată indică o valoare numerică unică utilizată pentru numele fişierului temporar. Dacă parametrul *uUnique* este diferit de zero, valoarea returnată specifică acelaşi număr. Dacă funcţia eşuează, valoarea returnată este zero.

Funcţia *GetTempFileName* creează un nume de fişier temporar de forma *cale\preuuuu.TMP*, unde *cale* reprezintă calea specificată de parametrul *lpPathName*, *pre* – primele trei caractere ale şirului *lpPrefixString*, iar *uuuu* valoarea hexazecimala a parametrului *uUnique*. Când Windows închide sesiunea de lucru, fişierele temporare al căror nume au fost create cu *GetTempFileName* nu sunt în mod automat şterse.

Dacă *uUnique* este zero, *GetTempFileName* încearcă să formeze un număr unic bazat pe ora curentă a sistemului. Dacă un fişier cu numele care rezultă există deja, numărul este incrementat cu unu şi testul este repetat. Testul continuă până când este găsit un nume unic de fişier. Atunci funcţia *GetTempFileName* creează un fişier cu acest nume unic şi îl închide. Când *uUnique* nu este zero, nu are loc nici o încercare de creare şi deschidere a fişierului.

CD-ROM-ul care însoţeşte cartea de faţă conţine programul *Create_Temp_cpp* care utilizează ambele funcţii *GetTempPath* şi *GetTempFileName* pentru obţinerea unui nume de fişier temporar. Când programul porneşte, el creează un fişier temporar pentru a fi utilizat în program şi afişează într-o casetă de mesaj numele fişierului temporar. Când programul se încheie, el închide şi şterge fişierul temporar.

PREZENTAREA FUNCŢIEI CREATENAMEDPIPE

C/C++1481

Programele dumneavoastră pot utiliza canale de transfer (utilizate în general în cadrul comunicării între două sau mai multe calculatoare) într-un mod asemănător intrărilor şi ieşirilor în fişiere. Dacă doriţi să utilizaţi un canal de transfer în program, el trebuie mai întâi creat. Funcţia *CreateNamedPipe* creează o instanţă a unui canal de transfer nominal şi returnează un identificator pentru operaţii ulterioare de tip canal de transfer. Un proces server canal de transfer nominal utilizează această funcţie fie pentru crearea primei instanţe a unui anumit canal de transfer nominal şi stabilirea atributelor sale de bază, fie pentru crearea unei noi instanţe a unui nou canal de transfer nominal existent. Veţi implementa funcţia *CreateNamedPipe* potrivit prototipului prezentat mai jos:

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName, // pointer la numele canalului
    DWORD dwOpenMode,       // mod de deschidere canal
```

```
    DWORD dwPipeMode,          // moduri specifice de canal
    DWORD nMaxInstances,       // numar maxim de instante
    DWORD nOutBufferSize,      // dimensiune buffer de iesire, in
                               // octeti
    DWORD nInBufferSize,       // dimensiune buffer de intrare, in
                               // octeti
    DWORD nDefaultTimeOut,     // timp de intrerupere, in
                               // milisecunde
    LPSECURITY_ATTRIBUTES lpSecurityAttributes //pointer la
// structura cu atributele de securitate
);
```

Funcția *CreateNamedPipe* acceptă parametrii prezentați în Tabelul 1461.1.

Parametru	Descriere
lpName	Indică un șir terminat cu NULL care identifică în mod unic canalul de transfer. Șirul trebuie să aibă următoarea formă: `\\.canal\numecanal`. Secțiunea *numecanal* poate cuprinde orice caracter definit de *backslash*, inclusiv numere și caractere speciale. Întregul nume al canalului nu poate depăși 256 caractere. Aceste nume nu depind de majuscule sau minuscule.
dwOpenMode	Specifică modul de acces la canalul de transfer, modul suprapus, modul scriere de transfer și modul acces de securitate ale identificatorului. Acest parametru trebuie să specifice unul din indicatoarele descrise în Tabelul 1481.2 și trebuie să specifice același indicator de mod pentru fiecare instanță a canalului.
dwPipeMode	Specifică tipul și modurile de citire sau scriere ale identificatorului de canal. Parametrul *dwPipeMode* trebuie să specifice unul sau mai multe indicatoare cu tipurile de mod descrise în Tabelul 1481.3 și să specifice același tip de mod pentru fiecare instanță a canalului de transfer. Dacă specificați zero, parametrul preia valoarea implicită a tipului octet de mod.
nMaxInstances	Specifică numărul maxim de instanțe care pot fi create pentru acest canal de transfer. Parametrul *nMaxInstances* trebuie să specifice același număr pentru toate instanțele. Valorile acceptabile sunt în intervalul de la 1 la *PIPE_UNLIMITED_INSTANCES*. Dacă parametrul este *PIPE_UNLIMITED_INSTANCES*, numărul de instanțe ale canalului de transfer ce pot fi create este limitat numai de disponibilitatea resurselor sistemului.
nOutBufferSize	Specifică numărul de octeți pentru a fi rezervați în bufferul de ieșire.
nInBufferSize	Specifică numărul de octeți pentru a fi rezervați în bufferul de intrare.
nDefaultTimeOut	Specifică valoarea de întrerupere implicită, în milisecunde, în cazul în care funcția *WaitNamedPipe* specifică *NMPWAIT_USE_DEFAULT_WAIT*. Fiecare instanță a canalului de transfer trebuie să specifice aceeași valoare.

Parametru	Descriere
lpSecurityAttributes	Pointer la o structură *SECURITY_ATTRIBUTES* care specifică descriptorul de securitate pentru un nou canal de transfer nominal și stabilește dacă procesele copil pot moșteni identificatorul returnat. Dacă *lpSecurityAttributes* este NULL, canalul de transfer nominal obține descriptorul de securitate implicit, iar identificatorul nu poate fi moștenit de procesele copil.

***Tabelul 1481.1** Parametrii funcției **CreateNamedPipe**.*

Așa cum ați aflat din Tabelul 1481.1, există mai multe valori de constante predefinite pe care Windows vă permite să le utilizați pentru parametrul *dwOpenMode*. Tabelul 1481.2 prezintă valorile modurilor de acces posibile.

Mod	Descriere
FILE_FLAG_WRITE_THROUGH	Este activat modul de scriere prin transfer. Acest mod afectează numai operațiile de scriere de tip octet la canalele de transfer și numai când procesele server și client se află pe calculatoare diferite. Dacă acest mod este activat, funcțiile care scriu la un canal de transfer nu se returnează până când datele scrise nu sunt transmise în rețea și nu sunt în bufferul canalului de transfer de pe calculatorul de la distanță. Dacă acest mod nu este activat, sistemul extinde eficiența operațiunilor în rețea prin reținerea în buffer a datelor până când se acumulează un număr minim de octeți sau până se epuizează timpul maxim.
FILE_FLAG_OVERLAPPED	Este activat modul suprapus. Dacă este activat acest mod, funcțiile care efectuează operațiuni de citire, scriere și conectare ce necesită un timp semnificativ pentru a se încheia, se pot returna imediat. Acest mod activează firul care a inițiat operația să efectueze alte operații, în timp ce operațiunile consumatoare de timp se execută în fundal. De exemplu, în modul suprapus, un fir poate gestiona operații de intrare/ieșire simultane (I/O) pe mai multe instanțe ale unui canal de transfer sau pot executa operații simultane de citire/scriere pe același identificator al canalului de transfer. Dacă funcția *CreateNamedPipe* nu activează modul suprapus, funcțiile care efectuează operațiile de citire, scriere și conectare la identificatorul canalului de transfer nu se vor returna până când operația nu este încheiată. Funcțiile *ReadFileEx* și *WriteFileEx* pot fi utilizate numai cu un identificator de canal în modul suprapus. Funcțiile *ReadFile, WriteFile, ConnectNamedPipe* și *TransactNamedPipe* se pot executa fie sincron, fie ca operații suprapuse. Acest parametru poate conține orice combinație a indicatoarelor cu modurile de acces de securitate prezentate în acest tabel. Aceste moduri pot fi diferite pentru diferite instanțe ale aceluiași canal de transfer. Puteți să le specificați indiferent de celelalte moduri specificate deja de *dwOpenMode*.

(continuare)

Mod	Descriere
WRITE_DAC	Procesul apelant va avea acces la scriere în lista de control discreționar al accesului (ACL) la canalul de transfer nominal.
WRITE_OWNER	Procesul apelant va avea acces la proprietarul canalului de transfer nominal.
ACCESS_SYSTEM_SECURITY	Procesul apelant va avea acces la scriere în sistemul ACL al canalului de transfer nominal.

Tabelul 1481.2 *Valorile posibile ale parametrului* ***dwOpenMode***.

Parametrul *dwPipeMode* vă permite specificarea modului în care vreți să utilizați instanța curentă a canalului de transfer. Puteți indica tipul octet, modul citire și modul scriere pentru acest parametru. Tabelul 1481.3 prezintă valorile posibile ale modurilor pentru parametrul *dwPipeMode*.

Mod	Descriere
PIPE_TYPE_BYTE	Datele sunt scrise la canalul de transfer sub forma unui flux de octeți. Acest mod nu poate fi utilizat cu *PIPE_READMODE_MESSAGE*.
PIPE_TYPE_MESSAGE	Datele sunt scrise la canalul de transfer sub forma unui flux de mesaje. Acest mod poate fi utilizat fie cu *PIPE_READMODE_MESSAGE*, fie cu *PIPE_READMODE_BYTE*.
PIPE_READMODE_BYTE	Datele sunt citite de la canalul de transfer sub forma unui flux de octeți. Acest mod poate fi utilizat fie cu *PIPE_TYPE_MESSAGE*, fie cu *PIPE_TYPE_BYTE*. În diferitele instanțe ale aceluiași canal de transfer, puteți specifica diferite moduri de citire. Dacă specificați zero, parametrul preia ca implicit modul citire-octeți.
PIPE_READMODE_MESSAGE	Datele sunt citite de la canalul de transfer sub forma unui flux de mesaje. Acest mod poate fi utilizat numai dacă este specificat de asemenea *PIPE_TYPE_MESSAGE*. În diferitele instanțe ale aceluiași canal de transfer, puteți specifica diferite moduri de citire. Dacă specificați zero, parametrul preia ca implicit modul citire-octeți.
PIPE_WAIT	Este activat modul cu blocare. Când e specificat identificatorul de canal în funcțiile *ReadFile, WriteFile sau ConnectNamedPipe* operațiile nu sunt încheiate până când funcția nu citește date, scrie toate datele sau conectează un client. Utilizarea modului *PIPE_WAIT* poate însemna în anumite situații așteptări nedefinte pentru efectuarea unei acțiuni de către un proces client. În diferitele instanțe ale aceluiași canal de transfer, puteți specifica diferite moduri de așteptare. Dacă specificați zero, parametrul preia ca implicit modul cu blocare.
PIPE_NOWAIT	Este activat modul fără blocare. În acest mod, funcțiile *ReadFile*, *WriteFile* și *ConnectNamedPipe* se returnează imediat.

Mod	Descriere
PIPE_ACCESS_DUPLEX	Canalul de transfer este bidirecțional. Atât procesele server, cât și cele client pot citi de la canalul de transfer și scrie în acesta. Acest mod oferă serverului echivalentul modului de acces *GENERIC_READ* | *GENERIC_WRITE* la canalul de transfer. Clientul poate specifica *GENERIC_READ* sau *GENERIC_WRITE*, sau ambele, când se conectează la canal folosind funcția *CreatePipe*. Puteți specifica diferite moduri de acces pentru instanțe diferite ale aceluiași canal de transfer.
PIPE_ACCESS_INBOUND	Fluxul de date în canalul de transfer este orientat exclusiv de la client spre server. Acest mod oferă serverului echivalentul modului de acces *GENERIC_READ* la canalul de transfer. Clientul trebuie să specifice modul de acces *GENERIC_WRITE* la conectarea cu respectivul canal de transfer.
PIPE_ACCESS_OUTBOUND	Fluxul de date în canalul de transfer este orientat exclusiv de la server la client. Acest mod oferă serverului echivalentul modului de acces *GENERIC_WRITE* la canalul de transfer. Clientul trebuie să specifice modul de acces *GENERIC_READ* la conectarea cu respectivul canal de transfer.

Tabelul 1481.3 *Valorile posibile pentru parametrul* ***dwPipeMode***.

Dacă funcția *CreateNamedPipe* reușește, valoarea returnată va fi identificatorul instanței de la extremitatea server a unui canal de transfer nominal. Dacă funcția eșuează, valoarea returnată va fi *INVALID_HANDLE_VALUE*. Pentru informații suplimentare, apelați *GetLastError*. Funcția returnează *ERROR_INVALID_PARAMETER* dacă *nMaxInstances* este mai mare decât *PIPE_UNLIMITED_INSTANCES*.

Pentru crearea unei instanțe a unui canal de transfer nominal cu funcția *CreateNamedPipe*, utilizatorului trebuie să-i fie permis accesul *FILE_CREATE_PIPE_INSTANCE* la obiectul canal de transfer nominal. Dacă funcția urmează să creeze un nou canal de transfer, lista de control a accesului (ACL), din parametrul cu atributele de securitate, definește controlul de acces discreționar pentru canalul de acces nominal.

Toate instanțele unui canal de transfer trebuie să specifice același tip de canal de transfer (tip-octet sau tip-mesaj), același mod de acces (*duplex*, *inbound* sau *outbound*), aceeași contorizare a instanțelor și aceeași valoare a intervalului de întrerupere. Dacă instanțele utilizează valori diferite, această funcție eșuează, iar *GetLastError* returnează *ERROR_ACCESS_DENIED*.

Dimensiunile bufferelor de intrare de intrare și ieșire sunt orientative. Dimensiunea efectivă rezervată pentru fiecare extremitate a canalului de transfer nominal este valoarea implicită dată de sistem, valoarea minimă sau maximă a sistemului sau dimensiunea rotunjită la următoarea limită de alocare. Serverul canalului nu ar trebui să efectueze o operație de blocare a citirii până când nu pornește canalul client. Altfel, apare o condiție de cursă. O condiție de cursă apare de regulă atunci când codul de inițializare (cum ar rutina de pornire a unui proces) trebuie să blocheze și să examineze identificatorii moșteniți. Condiția de cursă este o eroare într-un proces multifir, unde codul unui fir se corelează cu cel al unui al doilea fir pentru a îndeplini anumite acțiuni, dar fără să aibă loc nici un fel de sincronizări între fire. Procesul funcționează dacă al doilea fir „câștigă" cursa prin îndeplinirea acțiunii

sale, înainte ca primul fir să aibă nevoie de al doilea fir. Procesul eşuează dacă primul fir „câştigă" cursa.

Programul trebuie întotdeauna să şteargă instanţa unui canal de transfer nominal când ultimul identificator al instanţei canalului de transfer nominal a fost închis. În secţiunea 1483, veţi utiliza funcţiile *CallNamedPipe* şi *ConnectNamedPipe* împreună cu *CreateNamedPipe* pentru a executa o ieşire pe un server de reţea.

1482 CONECTAREA UNUI CANAL DE TRANSFER



Aşa cum aţi învăţat în secţiunea 1481, programele dumneavoastră pot utiliza funcţia *CreateNamedPipe* pentru a crearea unui canal de transfer nominal sau conectarea la acesta. De multe ori însă, programul dumneavoastră va rula pe o maşină client, care forţează conectarea la un canal de transfer nominal pentru asigurarea comunicării cu serverul, în loc de a crea un canal de transfer nominal (cum se întâmplă cu serverul canalului). În plus, înainte ca programul să se conecteze la canalul de transfer al serverului, canalul de transfer nominal trebuie să apeleze funcţia *ConnectNamedPipe* care face posibil ca un proces server al canalului de transfer nominal să aştepte un proces client pentru conectarea la o instanţă a canalului de transfer nominal. Un proces client se conectează la instanţă, apelând fie funcţia *CreateFile*, fie *CallNamedPipe* (despre aceasta din urmă veţi învăţa în secţiunea 1483). Veţi implementa funcţia *ConnectNamedPipe* potrivit prototipului prezentat mai jos:

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,           // identificator pentru canalul
                                 // de transfer pt. conectare
    LPOVERLAPPED lpOverlapped    // pointer la o structura de
                                 // suprapunere
);
```

Parametrul *hNamedPipe* identifică extremitatea server a unei instanţe a canalului de transfer nominal. Acest identificator este returnat de funcţia *CreateNamedPipe*. Parametrul *lpOverlapped* indică o structură *OVERLAPPED* (care, aşa cum aţi învăţat în secţiunea 1451, vă permite operaţiuni I/O asincrone). Dacă funcţia reuşeşte, valoarea returnată este diferită de zero. Dacă funcţia eşuează, valoarea returnată este zero. Pentru informaţii suplimentare despre eroare, apelaţi *GetLastError*.

Un proces server al canalului de transfer nominal poate utiliza *ConnectNamedPipe* cu o instanţă de canal nou creată. El poate, de asemenea, fi utilizat cu o instanţă anterior conectată la un alt proces client; în acest caz, procesul server trebuie pentru început să apeleze funcţia *DisconnectNamedPipe* pentru deconectarea identificatorului de la clientul anterior, înainte ca identificatorul să poată fi reconectat la noul client. În caz contrar, *ConnectNamedPipe* returnează *FALSE şi GetLastError* returnează *ERROR_NO_DATA*, când clientul anterior şi-a închis identificatorul sau *ERROR_PIPE_CONNECTED* dacă nu şi-a închis identificatorul.

Comportamentul funcţiei *ConnectNamedPipe* depinde de două condiţii: dacă modul de aşteptare al identificatorului de canal este stabilit cu blocare sau fără blocare şi dacă funcţia este stabilită să se execute în mod sincron sau în mod suprapus. Serverul specifică iniţial modul de aşteptare al identificatorului de canal de transfer în funcţia *CreateNamedPipe* şi poate fi schimbat cu funcţia *SetNamedPipeHandleState*.

Dacă *hNamedPipe* a fost deschis cu *FILE_FLAG_OVERLAPPED*, parametrul *lpOverlapped* nu trebuie să fie NULL. El trebuie să indice o structură *OVERLAPPED* validă. Dacă *hNamedPipe* a fost deschis cu *FILE_FLAG_OVERLAPPED* şi *lpOverlapped* este NULL, funcţia poate raporta incorect că operaţiunea de conectare este îndeplinită. Dacă *hNamedPipe* a fost creat cu *FILE_FLAG_OVERLAPPED* şi *lpOverlapped* nu este NULL, structura *OVERLAPPED* indicată de *lpOverlapped* trebuie să conţină un identificator de obiect eveniment cu iniţializare manuală (pe care serverul îl poate crea cu funcţia *CreateEvent*).

Dacă funcţia *CreateNamedPipe* nu deschide identificatorul *hNamedPipe* cu *FILE_FLAG_OVERLAPPED*, iar *lpOverlapped* este NULL, funcţia nu se returnează până când un client nu este conectat sau nu apare o eroare. Operaţiunile de sincronizare reuşite rezultă atunci când funcţia returnează TRUE, în cazul în care un client se conectează după ce funcţia a fost apelată. Când clientul se conectează înainte ca funcţia să fie apelată, funcţia returnează *FALSE*, iar *GetLastError* returnează *ERROR_PIPE_CONNECTED*. Aceasta se poate întâmpla dacă un client se conectează în intervalul dintre apelul la *CreateNamedPipe* şi apelul la *ConnectNamedPipe*. În această situaţie, conectarea între client şi server este operativă chiar dacă funcţia returmează *FALSE*.

Dacă *hNamedPipe* nu a fost deschis cu *FILE_FLAG_OVERLAPPED* şi *lpOverlapped* nu este NULL, operaţia se execută asincron. Funcţia se returnează imediat cu valoarea de returnare *FALSE*. Dacă un proces client se conectează înainte ca funcţia să fie apelată, *GetLastError* returnează *ERROR_PIPE_CONNECTED*. Altfel, *GetLastError* returnează valoarea *ERROR_IO_PENDING*, ceea ce arată că operaţia se execută în fundal. În această situaţie, obiectul eveniment din structura *OVERLAPPED* este fixat pe starea nesemnalizat, înainte ca funcţia *ConnectNamedPipe* să se returneze şi este fixat pe starea semnalizat când un client se conectează la această instanţă a canalului de transfer.

Procesul server poate utiliza orice funcţie de aşteptare sau *SleepEx* pentru a stabili când starea obiectului eveniment este semnalizată, ca apoi să poată utiliza funcţia *GetOverlappedResult* pentru a obţine rezultatele operaţiei *ConnectNamedPipe*. Dacă identificatorul de canal de transfer specificat este în modul fără blocare, funcţia *ConnectNamedPipe* se va returna întotdeauna imediat. În modul fără blocare, *ConnectNamedPipe* returnează *TRUE* când este prima dată apelată pentru instanţa unui canal deja deconectat de la clientul anterior. Faptul precizează că acel canal de transfer este acum disponibil pentru conectarea lui la un nou proces client. În toate celelalte situaţii, în care identificatorul de canal se află în modul fără blocare, *ConnectNamedPipe* returnează *FALSE*. În aceste situaţii, *GetLastError* returnează *ERROR_PIPE_LISTENING*, dacă nu este conectat nici un client, *ERROR_PIPE_CONNECTED* dacă este conectat un client şi *ERROR_NO_DATA*, dacă un client anterior şi-a închis propriul identificator de canal, dar serverul nu a fost deconectat. Reţineţi că, atunci când un canal este în modul fără blocare, o bună conectare între client şi server există numai după ce este recepţionată eroarea *ERROR_PIPE_CONNECTED*.

Observaţie: *Funcţia ConnectNamedPipe acceptă modul fără blocare pentru a asigura compatibilitatea cu Microsoft LAN Manager 2.0. Nu e recomandabil să utilizaţi această funcţie pentru realizarea intrărilor şi ieşirilor (I/O) asincrone prin intermediul canalelor de transfer nominale.*

1483 APELAREA UNUI CANAL DE TRANSFER NOMINAL



Programele dumneavoastră pot crea canale de transfer nominale la extremitatea server a unei conexiuni în rețea. Extremitatea client trebuie să se conecteze la canalul de transfer pentru transmiterea informațiilor. În cadrul programelor, puteți utiliza fie *CreateFile*, fie *CallNamedPipe* pentru conectarea la un canal de transfer. Funcția *CallNamedPipe* realizează conexiunea la un canal de transfer de tip mesaj (și așteaptă dacă o instanță a canalului nu este disponibilă), efectuează operații de citire și scriere din și, respectiv, la canalul de transfer, apoi procedează la închiderea canalului. Veți implementa funcția *CallNamedPipe* potrivit prototipului prezentat mai jos:

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,  // pointer la numele fisierului
    LPVOID lpInBuffer,        // pointer la bufferul de scriere
    DWORD nInBufferSize,      // dimensiunea bufferului de
                              // scriere, in octeti
    LPVOID lpOutBuffer,       // pointer la bufferul de citire
    DWORD nOutBufferSize,     // dimensiunea bufferului de
                              // citire, in octeti
    LPDWORD lpBytesRead,      // pointer la numarul de octeti
                              // cititi
    DWORD nTimeOut            // timp de intrerupere, in
                              // milisecunde
);
```

Funcția *CallNamedPipe* acceptă parametrii prezentați în Tabelul 1483.1.

Parametru	Descriere
lpNamedPipeName	Pointer la un șir terminat cu NULL cu numele canalului de transfer.
lpInBuffer	Pointer la bufferul care conține datele scrise de *CallNamedPipe* în canalul de transfer.
nInBufferSize	Specifică dimensiunea, în octeți, a bufferului de scriere.
lpOutBuffer	Pointer la bufferul care primește datele citite de *CallNamedPipe* din canalul de transfer.
nOutBufferSize	Specifică dimensiunea, în octeți, a bufferului de citire.
lpBytesRead	Pointer la o variabilă reprezentată pe 32 de biți care primește numărul de octeți citiți de *CallNamedPipe* din canalul de transfer.
nTimeOut	Specifică numărul de milisecunde necesare pentru așteptarea disponibilizării canalului de transfer nominal. Pe lângă valorile numerice, programul dumneavoastră poate specifica valorile speciale enumerate în Tabelul 1483.2.

***Tabelul 1483.1** Parametrii acceptați de funcția **CallNamedPipe**.*

Când apelați un canal de transfer în rețea, este important să plasați o limită intervalului de timp în care programul poate aștepta răspunsul de la canalul de transfer. O limită de timp previne ca un calculator client să aștepte nedefinit conectarea unui canal de transfer ocupat sau chiar inexistent. Pe lângă specificarea numărului de milisecunde pentru așteptarea

răspunsului de la canalul de transfer nominal, puteţi utiliza valorile din Tabelul 1483.2 pentru controlul intervalului de timp necesar pentru ca procesele client să aştepte conectarea la un canal de transfer nominal.

Valoare	**Semnificaţie**
NMPWAIT_NOWAIT	Nu aşteaptă pentru canalul de transfer nominal. Dacă respectivul canal de transfer nu este disponibil, funcţia returnează o eroare.
NMPWAIT_WAIT_FOREVER	Aşteaptă nedefinit.
NMPWAIT_USE_DEFAULT_WAIT	Utilizează timpul de întrerupere implicit, specificat în apelul la funcţia *CreateNamedPipe*.

Tabelul 1483.2 *Constantele cu duratele de aşteptare posibile pentru funcţia* ***CallNamedPipe***.

Dacă funcţia reuşeşte, ea va returna o valoare diferită de zero. Dacă funcţia eşuează, ea va returna valoarea zero. Pentru informaţii suplimentare despre eroare, apelaţi funcţia *GetLastError*.

Apelarea lui *CallNamedPipe* este echivalentă cu apelarea funcţiilor *CreateFile* (sau *WaitNamedPipe*, când *CreateFile* nu poate deschide imediat canalul de transfer), *TransactNamedPipe* şi *CloseHandle*. Funcţia *CreateFile* este apelată cu indicatorul de acces *GENERIC_READ | GENERIC_WRITE*, un indicator FALSE pentru moştenirea identificatorului şi un mod de partajare zero (indicând absenţa partajării acestei instanţe a canalului de transfer). Dacă mesajul scris la canalul de transfer de către procesul server este mai lung decât *nOutBufferSize*, funcţia *CallNamedPipe* returnează FALSE şi *GetLastError* returnează *ERROR_MORE_DATA*. Restul mesajului este abandonat de procesul server deoarece *CallNamedPipe* închide identificatorul canalului de transfer înaintea returnării.

CD-ROM-ul care însoţeşte cartea de faţă conţine programele *Server.cpp* şi *Client.cpp*. După compilare şi executare, programul *Server.cpp* va crea un canal de transfer nominal pe maşina locală. Dacă apoi compilaţi şi executaţi programul *Client.cpp*, acest program va apela canalul de transfer nominal. Programul *Client.cpp* va afişa mesajul şi va scrie apoi un şir de date în canalul de transfer nominal. Când serverul va recepţiona mesajul de la canalul de transfer nominal, el va afişa informaţiile în cadrul propriului său proces. Remarcaţi că exemplul va rula pe o singură maşină care va fi atât server, cât şi client.

DECONECTAREA UNUI CANAL DE TRANSFER NOMINAL

C/C++1484

Programele dumneavoastră pot utiliza canale de transfer nominal pentru comunicarea între două procese care rulează pe maşini diferite (sau chiar pe aceeaşi maşină). Însă, după încheierea partajării informaţiilor între două procese, este important să închideţi canalul de transfer nominal astfel încât să nu încetinească reţeaua. În general, veţi închide identificatorul unui canal de transfer nominal, mai întâi la extremitatea client, apoi veţi proceda la deconectarea canalului nominal de la server. Funcţia *DisconnectNamedPipe* deconectează serverul unei instanţe de canal de transfer nominal de la un proces client. Veţi utiliza funcţia *DisconnectNamedPipe* ca în următorul prototip:

```
BOOL DisconnectNamedPipe( HANDLE hNamedPipe );
```

Parametrul *hNamedPipe* identifică o instanţă a canalului de transfer nominal. Acest identificator trebuie creat cu funcţia *CreateNamedPipe*. Dacă funcţia reuşeşte, ea va returna o

valoare diferită de zero. Dacă funcţia eşuează, ea va returna valoarea zero. Pentru informaţii suplimentare despre eroare, apelaţi funcţia *GetLastError*.

Dacă extremitatea client a unui canal de transfer nominal este deschisă, funcţia *DisconnectNamedPipe* forţează această extremitate a canalului să se închidă. Clientul primeşte o eroare la o nouă încercare de acces la canalul de transfer. Clientul forţat să se deconecteze de la canal prin *DisconnectNamedPipe* trebuie să folosească şi funcţia *CloseHandle* pentru a închide extremitatea canalului.

Când procesul server deconectează o instanţă a unui canal de transfer, datele necitite din canal nu se pierd dacă se apelează *FlushFileBuffers* care nu se returnează până când procesul client nu a citit toate datele. Procesul client trebuie să apeleze funcţia *DisconnectNamedPipe* pentru deconectarea unui identificator de canal de la clientul anterior, înainte ca identificatorul să poată fi conectat la un alt client prin utilizarea funcţiei *ConnectNamedPipe*.

1485 *PROCESAREA ASINCRONĂ*



Programele dumneavoastră pot stoca şi regăsi informaţii din fişierele diverselor dispozitive cu ajutorul ieşirilor sincrone şi asincrone. Intrările şi ieşirile asincrone permit programelor dumneavoastră să efectueze operaţii pe fişiere care nu sunt sincronizate. Însă, este important să înţelegeţi mai bine de ce puteţi avea nevoie de operaţii de intrare/ieşire asincrone.

Comparativ cu majoritatea celorlalte operaţii realizate de calculatorul dumneavoastră, operaţiile de intrare/ieşire pe dispozitive sunt unele dintre cele mai lente. Unitatea centrală de prelucrare (CPU) efectuează operaţiile aritmetice şi chiar desenarea pe ecran cu o viteză mult mai mare decât cea cu care citeşte sau scrie date într-un fişier sau într-o reţea. Intrările şi ieşirile asincrone vă permit utilizarea mai multor fire de execuţie pentru a-i indica sistemului de operare să citească sau să scrie pe un dispozitiv, în timp ce restul codului aplicaţiei dumneavoastră îşi continuă execuţia.

Pentru a înţelege mai bine cum aceste operaţii de I/O asincrone pot îmbunătăţi performanţele programului dumneavoastră, să presupunem că aveţi de proiectat o aplicaţie simplă de baze de date. Atunci când utilizatorul deschide o bază de date, aplicaţia dumneavoastră va citi conţinutul bazei de date în memorie, precum şi într-un fişier index. După ce utilizatorul selectează baza de date pentru a fi încărcată, aplicaţia trebuie să facă o pauză pentru a încărca toate datele în memorie (probabil afişând un cursor clepsidră în timpul acesta).

Însă, dacă utilizaţi operaţii de I/O asincrone, un program poate începe operaţiile de I/O pe disc, operaţie ce o va efectua controllerul de disc şi va lăsa unitatea centrală să efectueze altă sarcină independentă în acelaşi timp. Operaţiile de I/O asincrone vă permit să efectuaţi mai multe activităţi de I/O în acelaşi timp – sau să începeţi o activitate de I/O ne-critică şi să lăsaţi calculatorul să o finalizeze în timpul dedicat, fără a încetini viteza execuţiei programului. Pe măsură ce programele dumneavoastră devin mai complexe şi veţi citi sau veţi scrie o cantitate mai mare de date în şi din unitatea de hard-disc sau alt mediu de stocare, beneficiile operaţiilor asincrone de I/O vor creşte.

UTILIZAREA INTRĂRILOR ȘI IEȘIRILOR ASINCRONE

C/C++1486

Pentru a accesa în mod asincron un dispozitiv, trebuie să apelați mai întâi funcția *CreateFile* pentru a deschide dispozitivul și să specificați indicatorul *FILE_FLAG_OVERLAPPED* ca parametru *dwFlagsAndAttrs*. Indicatorul *FILE_FLAG_OVERLAPPED* indică sistemului că intenționați să utilizați dispozitivul pentru operații de I/O asincrone.

Sistemul de operare Win32 vă permite utilizarea a patru tehnici diferite pentru efectuarea operațiilor asincrone de I/O. Cele patru tehnici provin dintr-o teorie de operare comună. Atunci când efectuați operații de I/O asincrone, mai întâi trebuie să emiteți o cerere de I/O către sistemul de operare. Sistemul de operare va depozita toate cererile de I/O în coada de așteptare și le va manevra intern. În timp ce sistemul de operare manevrează cererile de I/O, el permite întoarcerea firelor dumneavoastră de execuție, pentru a-și continua procesarea. La un anumit moment după aceea, sistemul de operare va încheia operația de I/O și va indica aplicației dumneavoastră că a trimis și a recepționat datele sau că a apărut vreo eroare.

Așa cum am observat anterior, există patru tehnici diferite de I/O. Tabelul 1486 le prezintă în ordinea complexității lor, de la cea mai ușor de înțeles și de implementat (semnalarea unui identificator de dispozitiv) până la cea mai dificilă (porturile de completare I/O).

Tehnică	Explicație
Semnalarea unui obiect dispozitiv	Utilizează o funcție *Wait* și un identificator de dispozitiv pentru a efectua operații de I/O asincrone. Nu este utilă dacă intenționați să efectuați mai multe cereri simultane de I/O asupra unui singur dispozitiv. Permite unui fir de execuție să emită o cerere de I/O și unui alt fir să prelucreze cererea.
Semnalarea unui obiect eveniment	Utilizează o funcție *WaitForMultipleObjects* și unul sau mai multe obiecte *event* pentru a efectua operații de I/O asincrone. Permite unui fir de execuție să emită o cerere de I/O și unui alt fir să prelucreze cererea.
I/O cu alertă	Utilizează o coadă de mesaje specială pentru a prelucra notificările sistemului de operare generate atunci când o operație de I/O asincronă a fost îndeplinită. Oferă o mai mare flexibilitate decât utilizarea unui obiect kernel eveniment, deoarece puteți utiliza funcții callback și puteți prelucra mesajele specifice pentru a răspunde informațiilor date de sistemul de operare în legătură cu acțiunea de I/O. Firul care a emis cererea de I/O trebuie, de asemenea, să prelucreze răspunsul. Puteți utiliza operațiile de I/O cu alertă numai în sistemele Windows NT.

(continuare)

Tehnică	Explicație
Porturile de completare I/O	Utilizează un *model de fire de execuție concurente* (explicat în secțiunea 1495) pentru a răspunde unui număr mare de cereri simultane de I/O. Permite unui fir de execuție să emită o cerere de I/O și unui alt fir să prelucreze această cerere. Porturile de completare, fiind o tehnică cu o mare scalabilitate, este utilizată îndeosebi de către programatorii profesioniști. Puteți utiliza porturile de completare I/O numai în sistemele Windows NT.

***Tabelul 1486** Cele patru tehnici de procesare asincronă a I/O.*

1487 *STRUCTURA OVERLAPPED*



După cum arată tabelul 1486, cea mai simplă tehnică pentru efectuarea de operații asincrone de I/O pe dispozitive este utilizarea semnalărilor date de identificatorii de dispozitive, pe care le veți utiliza în secțiunea 1488. Pentru a emite cererea de I/O, veți utiliza funcțiile *ReadFile* și *WriteFile* prezentate în secțiunile 1456 și 1457. Însă, pentru a efectua operații de I/O asincrone, programele dumneavoastră trebuie să transmită adresa unei structuri *OVERLAPPED* ca parametru *lpOverlapped*. Interfața API Win32 definește structura *OVERLAPPED* ca mai jos:

```
typedef struct _OVERLAPPED {
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED;
```

Tabelul 1487 prezintă membrii structurii *OVERLAPPED*.

Membru	Descriere
Internal	Specifică o stare dependentă de sistem. Acest membru este valid atunci când funcția *GetOverlappedResult* se returnează fără a stabili informația extinsă de eroare la *ERROR_IO_PENDING*. Rezervat pentru uzul sistemului de operare.
InternalHigh	Specifică lungimea datelor transferate. Acest membru este valid atunci când funcția *GetOverlappedResult* returnează *TRUE*. Rezervat pentru uzul sistemului de operare.
Offset	Specifică o poziție din fișier de la care să înceapă transferul. Poziția din fișier este deplasamentul de octeți pornind de la începutul fișierului. Procesul apelant stabilește acest membru înainte de a apela funcțiile *ReadFile* sau *WriteFile*. Procesul apelant ignoră acest membru atunci când citește sau scrie în canale de transfer nominale sau în dispozitive de comunicare.

Membru	Descriere
OffsetHigh	Conţine cuvântul mai semnificativ al deplasamentului de octeţi de la care să înceapă transferul. Procesul apelant ignoră acest membru atunci când citeşte sau scrie în canale de transfer nominale sau în dispozitive de comunicare.
hEvent	Identifică un eveniment fixat pe starea semnalată atunci când transferul s-a încheiat. Procesul apelant stabileşte acest membru înainte de apelarea funcţiilor *ReadFile*, *WriteFile*, *ConnectNamedPipe* sau *TransactNamedPipe*.

Tabelul 1487 *Membrii structurii* ***OVERLAPPED***.

Puteţi utiliza funcţia macro *HasOverlappedIoCompleted* pentru a determina dacă o operaţie de I/O asincronă a fost finalizată. Puteţi utiliza funcţia *CancelIo* pentru a abandona o operaţie de I/O asincronă.

OPERAŢIILE DE I/O ASINCRONE CU UN OBIECT KERNEL DE DISPOZITIV

C/C++1488

Programele dumneavoastră pot efectua operaţii de I/O asincrone utilizând patru tehnici diferite, dintre care cea mai simplă este utilizarea unui obiect kernel de dispozitiv. Atunci când efectuaţi operaţii de I/O asincrone asupra unui obiect kernel de dispozitiv, pur şi simplu indicaţi firului de execuţie să aştepte până când operaţia de I/O s-a încheiat. De exemplu, să presupunem că citiţi dintr-un fişier cu ajutorul funcţiei *ReadFile*, efectuaţi câteva prelucrări interimare, dar programul nu poate continua înainte de un anumit moment, adică până când operaţia de I/O nu se finalizează. În astfel de cazuri, puteţi construi codul în mod similar cu ceea ce arătăm în următorul fragment:

```
ReadFile(hFile, hBuffer, sizeof(hBuffer), &dwNumBytesRead,
   &Overlapped);
// urmeaza prelucrarile
WaitForSingleObject(hFile, INFINITE);
// asteapta pana cand toate datele se vor afla in buffer
```

Funcţia *WaitForSingleObject*, pe care o apelaţi împreună cu identificatorul pentru un dispozitiv de I/O asincrone, va aştepta până când sistemul de operare va încheia prelucrările corespunzătoare dispozitivului, înainte de a se elibera şi a permite firului să-şi continue execuţia. CD-ROM-ul ce însoţeşte această carte conţine programul *Simple_ASRead.cpp*, care utilizează tehnica obiectelor kernel de dispozitive pentru a efectua operaţii asincrone simple de I/O.

COTELE

C/C++1489

Atunci când programele dumneavoastră efectuează operaţii de I/O asincrone, sistemul de operare va menţine o listă a cererilor de I/O în curs. Sistemul de operare fixează dimensiunea listei la pornirea sistemului. Câteodată, o cerere de I/O asincronă poate eşua deoarece lista cererilor curente de I/O este deja plină. Dacă lista este plină atunci când emiteţi o altă cerere, *ReadFile* şi *WriteFile* vor returna valoarea *False*, iar *GetLastError* va returna fie *ERROR_INVALID_USER_BUFFER*, fie *ERROR_NOT_ENOUGH_MEMORY*. Mai mult, atunci când emiteţi o cerere de I/O, sistemul trebuie să „blocheze pagina" bufferului de

date al programului dumneavoastră. BUfferul de date al programelor dumneavoastră este parte a setului de lucru al programului și fiecare proces are un set de lucru maximal. Setul de lucru al unui proces este setul de pagini de memorie vizibile la acel moment de către proces, din memoria RAM fizică. Dacă nu aveți suficient spațiu în spațiul de lucru al procesului, emiterea unei cereri de I/O îl va face să eșueze, iar *GetLastError* va returna valoarea *ERROR_NOT_ENOUGH_QUOTA*. Puteți mări dimensiunea setului de lucru al programului dumneavoastră apelând funcția *SetProcessWorkingSize*, explicată în detaliu în secțiunea 1490.

1490 STABILIREA UNOR COTE MAI JOASE SAU MAI RIDICATE



După cum ați învățat în secțiunea 1489, programele dumneavoastră își pot mări dimensiunile setului lor de lucru (paginile de memorie vizibile la acel moment de către proces din memoria RAM fizică), atunci când necesită spațiu suplimentar în cadrul spațiului lor de lucru. Aceste pagini sunt rezidente și disponibile pentru utilizarea lor de către o aplicație, fără a declanșa o eroare de pagină. Dimensiunea setului de lucru al unui program este specificată în octeți. Dimensiunea minimă și cea maximă a setului de lucru afectează comportarea unui proces la paginarea memoriei virtuale. Funcția *SetProcessWorkingSetSize* stabilește dimensiunea minimă și cea maximă a setului de lucru pentru un proces specificat de dumneavoastră. În programele dumneavoastră veți utiliza funcția *SetProcessWorkingSetSize* după cum arătăm în următorul prototip:

```
BOOL SetProcessWorkingSetSize(
    HANDLE hProcess,                  // deschide identificatorul
                                      // catre proces
    DWORD dwMinimumWorkingSetSize,    // specifica dimensiunea
                                      // minima
    DWORD dwMaximumWorkingSetSize     // specifica dimensiunea
                                      // maxima
);
```

Parametrul *hProcess* este un identificator deschis pentru procesul căruia îi veți stabili dimensiunea setului de lucru. Sub Windows NT, identificatorul trebuie să aibă dreptul de acces *PROCESS_SET_QUOTA*. Parametrul *dwMinimumWorkingSetSize* specifică o dimensiune minimă a setului de lucru pentru un proces. Gestionarul memoriei virtuale încearcă să păstreze cel puțin această memorie rezidentă în proces, atunci când el este activ. Parametrul *dwMaximumWorkingSetSize* specifică o dimensiune maximă a setului de lucru pentru un proces. Gestionarul memoriei virtuale încearcă să nu păstreze mai mult decât această memorie rezidentă în proces, atunci când procesul este activ, iar memoria este deficitară. Dacă ambii parametri *dwMinimumWorkingSetSize* și *dwMaximumWorkingSetSize* au valoarea *0xFFFFFFFF*, funcția dimensionează la zero setul de lucru al procesului specificat. Acest lucru scoate procesul în afara memoriei RAM fizice. Dacă funcția reușește, ea va returna o valoare diferită de zero. Dacă funcția eșuează, ea va returna zero. Prin apelarea funcției *GetLastError* veți obține mai multe informații despre eroare.

Puteți elibera setul de lucru al programului prin specificarea valorii *0xFFFFFFFF*, atât ca dimensiune maximă, cât și minimă, pentru setul de lucru al procesului. Dacă valoarea fie a parametrului *dwMinimumWorkingSetSize*, fie a parametrului *dwMaximumWorkingSetSize* este mai mare decât dimensiunile curente ale setului de lucru al programului, procesul

respectiv trebuie să aibă privilegiul *SE_INC_BASE_PRIORITY_NAME*. Sistemul de operare alocă dimensiunile setului de lucru după principiul „primul venit, primul servit". De exemplu, dacă o aplicație stabilește cu succes 40Mb ca dimensiune minimă a setului său de lucru într-un sistem pe 64Mb, iar o a doua aplicație cere o dimensiune a setului de lucru de 40Mb, sistemul de operare respinge cererea celei de-a doua aplicații.

Utilizarea funcției *SetProcessWorkingSetSize* pentru a stabili dimensiunile minime și maxime ale setului de lucru al unei aplicații nu garantează că memoria cerută va fi rezervată sau că ea îi va rămâne rezidentă tot timpul. Atunci când aplicația este inactivă sau o situație de memorie insuficientă cauzează o cerere de memorie, sistemul de operare poate reduce setul de lucru al aplicației. O aplicație poate utiliza funcția *VirtualLock* pentru a bloca intervalele spațiului de adresă virtual din memorie al aplicației; însă, aceasta poate coborî performanțele programului dumneavoastră.

Atunci când măriți dimensiunea setului de lucru al unei aplicații, se reține memoria fizică din restul sistemului. Aceasta poate coborî performanțele altor aplicații și ale sistemului în ansamblu. De asemenea, poate duce la eșecuri ale operațiilor care necesită memorie fizică pentru a fi prezente; de exemplu, crearea de procese, fire de execuție și a unui grup kernel. Prin urmare, trebuie să utilizați cu atenție funcția *SetProcessWorkingSetSize*. Trebuie să analizați întotdeauna performanțele întregului sistem atunci când proiectați o aplicație.

FUNCȚIA GETLASTERROR

C/C++ 1491

După cum ați învățat în ultimele 40 de secțiuni, puteți adesea să obțineți mai multe informații despre o eroare de program. În general, programele dumneavoastră trebuie să apeleze funcția *GetLastError* pentru a obține aceste informații. Funcția *GetLastError* returnează valoarea codificată a ultimei erori a firului de execuție apelant. Sistemul de operare păstrează codul ultimei erori în concordanță cu firul de execuție. Nu se vor suprascrie codurile de eroare ale mai multor fire. În programele dumneavoastră, veți utiliza funcția *GetLastError* după modelul de mai jos:

```
DWORD GetLastError(void);
```

Funcția returnează codul ultimei erori a firului apelant. Funcțiile apelează funcția *SetLastError* pentru a stabili valoarea de cod a ultimei erori. Trebuie să apelați funcția *GetLastError* imediat ca valoarea de returnare a unei funcții indică faptul că un astfel de apel va avea ca rezultat date utile (cum ar fi informații extinse despre eroare) deoarece unele funcții vor apela *SetLastError(0)* atunci când se încheie cu succes, eliminând codul de eroare stabilit de cea mai recentă funcție eșuată.

Majoritatea funcțiilor din interfața Win32 API care stabilesc valoarea codificată a ultimei erori a firului fac acest lucru atunci când eșuează; numai câteva funcții o stabilesc atunci când se încheie cu succes. Un cod de eroare, cum ar fi *False*, *NULL*, *0xFFFFFFFF* sau *–1*, indică în general eșecul unei funcții. Codurile de eroare sunt valori pe 32 de biți (bitul 31 este cel mai semnificativ). Bitul 29 este rezervat pentru coduri de eroare definite de aplicații; nici un cod de eroare de sistem nu are acest bit 1. Dacă definiți un cod de eroare pentru aplicația dumneavoastră, fixați bitul 29 pe unu. Fixarea bitului 29 pe unu indică faptul că o aplicație a definit codul de eroare și se asigură că acest cod de eroare nu intră în conflict cu vreunul dintre codurile de eroare definite de sistem. Puteți utiliza funcția *FormatMessage* pentru a formata ieșirea generată de apelul funcției *GetLastError*. Secțiunea 1492 explică funcția *FormatMessage*.

CD-ROM-ul ce însoțește această carte cuprinde programul *GenerateError.cpp*. Atunci când compilați și executați programul *GenerateError.cpp*, el efectuează o serie de activități care vor genera fiecare câte o eroare, ce va fi afișată într-o casetă de mesaj.

1492 FORMATAREA MESAJELOR DE EROARE CU FORMATMESSAGE



După cum ați învățat în secțiunea 1491, programele dumneavoastră pot utiliza funcția *GetLastError* pentru a obține o reprezentare numerică a ultimei erori a unui fir. Însă dumneavoastră veți dori de obicei să vedeți reprezentarea în caractere a ultimei erori a firului.

Funcția *FormatMessage* formatează un șir de caractere mesaj. Funcția cere ca intrare o definiție a unui mesaj. Definiția mesajului poate proveni dintr-un buffer transmis funcției. El mai poate proveni și dintr-o resursă tabel de mesaje, dintr-un modul deja încărcat. În plus, procesul apelant poate cere funcției să caute în resursele cu tabelele de mesaje ale sistemului acea definiție a mesajului. Funcția găsește definiția mesajului într-un tabel de mesaje bazându-se pe un identificator de mesaj sau de limbă. Funcția copiază textul mesajului formatat într-un buffer de ieșire, prelucrând fiecare dintre secvențele inserate, dacă se cere. În programele dumneavoastră, veți utiliza funcția *FormatMessage* după cum arătăm în următorul prototip:

```
DWORD FormatMessage(
   DWORD dwFlags,        // optiuni de prelucrare si ale sursei
   LPCVOID lpSource,     // pointer la sursa mesajului
   DWORD dwMessageId,    // identificatorul mesajului cerut
   DWORD dwLanguageId,   // identificatorul limbajului pentru
                         // mesajul cerut
   LPTSTR lpBuffer,      // pointer la bufferul de mesaje
   DWORD nSize,          // dimensiunea maxima a bufferului de
                         // mesaje
   va_list *Arguments    // adresa matricei de inserari in mesaj
);
```

Tabelul 1492.1 prezintă parametrii funcției *FormatMessage*:

Parametrul	Descriere
dwFlags	Conține un set de indicatoare pe biți care specifică aspecte ale procesului de formatare și modul de interpretare al parametrului *lpSource*. Octetul mai puțin semnificativ al lui *dwFlags* specifică modul în care funcția manevrează întreruperile de linii din bufferul de ieșire. De asemenea, octetul mai puțin semnificativ poate specifica lățimea maximă a unei linii de ieșire formatate. Puteți specifica o combinație a indicatoarelor pe biți detaliate în tabelul 1491.2.
lpSource	Specifică locația definiției mesajului. Tipul acestui parametru depinde de valorile parametrului *dwFlags*. Dacă stabiliți *dwFlags* la *FORMAT_MESSAGE_FROM_HMODULE*, *lpSource* va fi un *hModule* al modulului ce conține tabelul de mesaje în care trebuie căutat. Pe de altă parte, dacă stabiliți *dwFlags* la *FORMAT_MESSAGE_FROM_STRING*, *lpSource* va fi un *LPSTR* care indică textul mesajului neformatat. Dacă nu stabiliți nici unul dintre aceste indicatoare în *dwFlags*, atunci funcția va ignora parametrul *lpSource*.

Parametrul	Descriere
dwMessageId	Specifică identificatorul de limbaj pe 32 de biți al mesajului cerut. Acest parametru este ignorat dacă *dwFlags* conține *FORMAT_MESSAGE_FROM_STRING*.
dwLanguageId	Specifică identificatorul de mesaj pe 32 de biți al mesajului cerut. Acest parametru este ignorat dacă *dwFlags* conține *FORMAT_MESSAGE_FROM_STRING*.
	Dacă transmiteți un *LANGID* specificat în acest parametru, *FormatMessage* va returna un mesaj numai pentru acel *LANGID*. Dacă funcția nu poate găsi un mesaj pentru acel *LANGID*, ea va returna eroarea *ERROR_RESOURCE_LANG_NOT_FOUND*.
lpBuffer	Pointer la un buffer pentru mesajul formatat (și terminat cu *NULL*). Dacă *dwFlags* cuprinde *FORMAT_MESSAGE_ALLOCATE_BUFFER*, funcția alocă un buffer cu ajutorul funcției *LocalAlloc* și plasează adresa bufferului la adresa specificată în *lpBuffer*.
nSize	Dacă nu stabiliți indicatorul *FORMAT_MESSAGE_ALLOCATE_BUFFER*, acest parametru specifică numărul maxim de octeți (în versiune ANSI) sau caractere (versiune Unicode) care poate fi depozitat în bufferul de ieșire. Dacă este stabilit *FORMAT_MESSAGE_ALLOCATE_BUFFER*, acest parametru specifică numărul minim de octeți sau caractere pentru a fi alocate bufferului de ieșire.
Argumente	Indică o matrice de valori pe 32 de biți care sunt utilizate ca valori de inserat în mesajul formatat. %1 din șirul de formatat indică prima valoare din matricea Argumente; %2 indică cel de-al doilea argument; și așa mai departe.

Tabelul 1492.1 *Parametrii funcției* ***FormatMessage***.

Interpretarea pe care o dă funcția fiecărei valori pe 32 de biți depinde de informația de formatat conținută în parametrul *dwFlags* și locația definiției mesajului real. În mod implicit, fiecare valoare este tratată ca un pointer către un șir de caractere terminat în *NULL*. În mod implicit, parametrul *Argumente* este de tipul *va_list**, care este un tip de date specific limbajului și implementării și care descrie un număr de argumente variabil. Dacă nu aveți un pointer de tipul *va_list**, atunci specificați indicatorul *FORMAT_MESSAGE_ARGUMENT_ARRAY* și transmiteți un pointer către o matrice de valori pe 32 de biți; aceste valori sunt intrări către mesaj, formatate ca valori de inserare. Fiecărei inserări trebuie să îi corespundă un element din matrice. Tabelul 1492.2 detaliază indicatoarele de format pentru parametrul *dwFlags*.

Valoare	Semnificație
FORMAT_MESSAGE_ALLOCATE_BUFFER	Specifică faptul că parametrul *lpBuffer* este un pointer la un pointer *PVOID* și că parametrul *nSize* specifică numărul minim de octeți (versiune ANSI) sau caractere (versiune Unicode) pentru a fi alocați bufferului mesajului de ieșire. Funcția alocă un buffer suficient de mare pentru a cuprinde mesajul formatat și plasează un pointer către bufferul alocat la adresa specificată de *lpBuffer*.

(continuare)

Valoare	Semnificație
FORMAT_MESSAGE_IGNORE_INSERTS	Specifică faptul că secvențele inserate în definiția mesajului vor fi ignorate și le transmite nemodificate către bufferul de ieșire. Acest indicator este util pentru a păstra un mesaj pentru o formatare ulterioară. Dacă acest indicator este activ, parametrul *Arguments* va fi ignorat.
FORMAT_MESSAGE_FROM_STRING	Specifică faptul că *lpSource* este un pointer către o definiție a unui mesaj terminată cu NULL. Definiția mesajului poate conține secvențe de inserat, așa cum pot și mesajele dintr-o resursă tabel de mesaje. Nu poate fi utilizat împreună cu indicatorul *FORMAT_MESSAGE_FROM_HMODULE* sau cu *FORMAT_MESSAGE_FROM_SYSTEM.*
FORMAT_MESSAGE_FROM_HMODULE	Specifică faptul că *lpSource* este un identificator de modul conținând resursa sau resursele de tabele de mesaje în care se va căuta. Dacă identificatorul *lpSource* este *NULL*, va fi căutat fișierul imagine al procesului curent. Nu poate fi utilizat împreună cu *FORMAT_MESSAGE_FROM_STRING.*
FORMAT_MESSAGE_FROM_SYSTEM	Specifică faptul că funcția trebuie să caute mesajul cerut în resursele tabelelor de mesaje ale sistemului. Dacă acest indicator este specificat împreună cu *FORMAT_MESSAGE_FROM_HMODULE*, funcția caută în tabelul de mesaje al sistemului, dacă mesajul nu este întâlnit în modulul specificat de *lpSource*. Nu poate fi utilizat cu *FORMAT_MESSAGE_FROM_STRING.*
FORMAT_MESSAGE_ARGUMENT_ARRAY	Specifică faptul că parametrul *Argumente* nu este o structură de tip *va_list**, ci numai un pointer către o matrice de valori pe 32 de biți care reprezintă argumentele.

Tabelul 1492.2 *Valori posibile pentru parametrul* ***dwFlags*** .

Dacă funcția se încheie cu succes, ea va returna numărul de octeți (versiune ANSI) sau de caractere (versiune Unicode) stocate în bufferul de ieșire, excluzând caracterul NULL de sfârșit. Dacă funcția eșuează, ea returnează zero. Pentru a obține mai multe informații asupra erorii, apelați *GetLastError.*

În cadrul textului mesajului, funcția acceptă câteva secvențe escape pentru formatarea dinamică a mesajului. Tabelul 1492.3 arată aceste secvențe escape și semnificația lor. Toate secvențele escape încep cu caracterul procent (%).

Secvență escape	Semnificație
%0	Încheie o linie a textului mesajului fără a insera un caracter de linie nouă. Această secvență *escape* poate fi utilizată pentru a construi linii lungi sau pentru a termina mesajul însuși fără a adăuga un caracter de linie nouă. Este util pentru mesaje de *prompt*.
%n!printf format string!	Identifică o inserare. Valoarea lui *n* se poate afla în intervalul de la 1 la 99. Șirul de formatare din *printf* (care trebuie încadrat cu semne de exclamare) este opțional și implicit este *!s!* dacă nu este specificat. Șirul de formatare din *printf* poate conține specificatorul * pentru precizia componentei sau lungimea ei. Dacă * este specificat pentru o componentă, funcția *FormatMessage* utilizează inserarea *%n+1*; ea utilizează *%n+2* dacă * este specificat pentru ambele componente. Funcția nu acceptă specificatorii *e, E, f* și *g* de format din *printf*. Alternativa este utilizarea funcției *sprintf* pentru a formata numărul în virgulă mobilă într-un buffer temporar, apoi utilizarea acelui bufferului ca șir de caractere de inserat.

Tabelul 1492.3 *Secvențe escape posibile pentru caracterele de formatare, precedate de %.*

Funcția formatează oricare alt caracter care nu este cifră, ce urmează unui semn procent în mesajul de ieșire ca pe caracterul însuși, fără semnul de procent. Tabelul 1492.4 prezintă câteva exemple de ieșiri ale unor caractere ce nu sunt de formatare.

Șir de formatare	Ieșire rezultată
%%	Un singur semn de procent în textul mesajului formatat.
%n	Întreruperea liniei cu reluarea pe un rând nou, atunci când șirul de formatat atinge sfârșitul liniei. Acest tip de formatare este util atunci când *FormatMessage* furnizează întreruperi de linii obișnuite, pentru ca mesajul să corespundă unei anumite dimensiuni.
%space	Un spațiu în textul mesajului formatat. Acest șir de formatare poate fi utilizat pentru a asigura un număr corespunzător de spații într-o linie de text.
%.	Un singur punct în textul mesajului formatat. Acest șir de formatare poate fi utilizat pentru a introduce un singur punct la începutul unei linii, fără a termina definiția textului mesajului.
%!	Un singur punct de exclamare în textul mesajului formatat. Acest șir de formatare poate fi utilizat pentru a introduce un semn de exclamare imediat după o inserare, fără a fi confundat cu începutul unui șir de formatare din *printf*.

Tabelul 1492.4 *Exemple de ieșiri ale unor caractere ce nu sunt de formatare.*

Programul *GenerateError.cpp* din secțiunea 1491 folosește din plin funcția *FormatMessage*.

OPERAȚII DE I/O ASINCRONE CU UN OBIECT KERNEL DE EVENIMENT

C/C++1493

În secțiunea 1490 ați utilizat funcția *WaitForSingleObject* împreună cu un identificator de dispozitiv asincron pentru a efectua operații de I/O asincrone. Atunci când lucrați cu un

obiect kernel de dispozitiv, așa cum detaliază secțiunea 1490, este relativ simplu și direct, dar în particular nu este util atunci când se manevrează simultan mai multe cereri de I/O. Dacă, de exemplu, încercați să efectuați simultan mai multe cereri de I/O asupra unui unic fișier, așteptarea identificatorului de fișier nu vă va ajuta deoarece va fi semnalat numai după ce primul eveniment se încheie și va trebui să așteptați din nou eliberarea sa, ceea ce poate avea ca efect o așteptare la infinit.

Puteți, de asemenea, să utilizați și funcția *CreateEvent* pentru a crea un obiect kernel de eveniment apoi veți putea identifica acest obiect în cadrul membrului *hEvent* al structurii *OVERLAPPED*, pe care o transmiteți funcției dumneavoastră de cerere de I/O asincrone (fie *ReadFile*, fie *WriteFile*). Atunci când transmiteți un eveniment într-o asemenea manieră, sistemul de operare va stabili automat evenimentul ca semnalat atunci când operația de I/O se încheie. Însă, din cauză că ele pot stabili un eveniment diferit pentru fiecare operație de I/O, programele dumneavoastră pot răspunde adecvat la încheierea unei anumite operații de I/O și nu a unei alte operații.

De fiecare dată când efectuați o operație de I/O asincronă, programul dumneavoastră trebuie să creeze un nou eveniment pentru acea operație. Prin aceasta, de fiecare dată când sistemul de operare își încheie acțiunea, el va stabili evenimentul operației apelante în stare semnalată. Așa cum este descris în secțiunea 1494, puteți apoi aștepta evenimentele pe care le doriți să se finalizeze.

1494 UTILIZAREA FUNCȚIEI WAITFORMULTIPLEOBJECTS CU OPERAȚII DE I/O ASINCRONE



Puteți utiliza funcția *WaitForMultipleObjects* pentru a aștepta apariția a unul sau mai multor evenimente sau apariția unui anume subset de evenimente. Atunci când efectuați operații de I/O asincrone, trebuie să utilizați funcția *WaitForMultipleObjects* pentru a sincroniza firele dumneavoastră de execuție cu un anumit set de evenimente. Programele dumneavoastră trebuie să apeleze *WaitForMultipleObjects* împreună cu identificatorii tuturor evenimentelor pe care sistemul de operare trebuie să le finalizeze înainte ca programul dumneavoastră să își poată continua activitatea și apoi să aștepte ca acele evenimente să intre în starea de semnalare. În general, veți efectua astfel de acțiuni utilizând fragmente de cod similare celui de mai jos:

```
Eveniment[1] = HANDLE CreateEvent(LPSECURITY_ATTRIBUTES
                                  lpEventAttributes, BOOL
                                  bManualReset, BOOL
                                  bInitialState, LPCTSTR
                                  Eveniment1);
Overlapped1.hEvent = Eveniment[1];
ReadFile(hFile, bBuffer, sizeof(bBuffer),&dwNumBytesRead,
      &Overlapped1);
Eveniment[2] = HANDLE CreateEvent(LPSECURITY_ATTRIBUTES
                                  lpEventAttributes, BOOL
                                  bManualReset, BOOL
                                  bInitialState, LPCTSTR
                                  Eveniment2);
```

```
Overlapped2.hEvent = Eveniment[2];
ReadFile(hFile, bBuffer, sizeof(bBuffer),&dwNumBytesRead,
         &Overlapped2);
// prelucrari suplimentare
DWORD WaitForMultipleObjects(2, CONST HANDLE *Event, BOOL
                             bWaitAll, INFINITE);
```

Fragmentul de cod creează un eveniment și transmite acel eveniment către prima acțiune de citire. Apoi, fragmentul de cod creează un al doilea eveniment și îl transmite către cea de-a doua acțiune de citire. În final, codul așteaptă cele două evenimente se returnează înainte de a-și continua procesările.

Obiectele kernel de eveniment sunt foarte utile în gestionarea operațiilor de I/O asincrone. Pericolul ce poate apărea atunci când lucrați cu obiecte kernel de eveniment este de a stabili un obiect kernel de eveniment ca eveniment cu *auto-reset*, deoarece este posibil ca un fir de execuție să rămână „agățat" la infinit așteptând ca evenimentul cu *auto-reset* să se inițializeze, chiar dacă funcția a finalizat anterior operația de I/O. Dacă apelați *GetOverlappedResult* pentru a determina câți octeți a transferat cu succes operația de I/O, funcția va reinițializa evenimentul în starea de nesemnalat. Pe scurt, observați cu atenție secvența de funcții pe care le efectuați atunci când utilizați obiecte kernel de eveniment pentru a manevra operațiile de I/O asincrone.

PREZENTAREA PORTURILOR DE INTRARE/IEȘIRE DE COMPLETARE

C/C++1495

Cea de-a patra tehnică pe care programele dumneavoastră o pot utiliza pentru efectuarea de operații de I/O asincrone este utilizarea porturilor de I/O de completare. Veți utiliza în general porturile I/O de completare atunci când veți proiecta un program ce va servi sute sau chiar mii de utilizatori (cum ar fi un server Web). Porturile I/O de completare sunt extrem de sigure și puternice și pot manevra în siguranță un număr mare de activități de comunicare. Atunci când creați o aplicație de service, veți face aceasta într-unul din cele două moduri:

- În *modelul serial*, un singur fir așteaptă ca un client să efectueze o cerere (în general, prin intermediul rețelei). Atunci când cererea este primită, firul intră în acțiune și prelucrează cererea clientului.
- În *modelul concurent*, un singur fir așteaptă cererea unui client și apoi creează un nou fir pentru a prelucra acea cerere. În timp ce noul fir prelucrează cererea clientului, firul inițial ciclează din nou, așteptând o altă cerere client. Atunci când firul ce tratează cererea clientului își încheie procesarea, firul „moare".

Modelul serial este un model limitat, prin aceea că nu tratează bine mai multe cereri simultane (deoarece numai un singur fir tratează cererile). Din contră, modelul concurent este capabil de a trata un număr extrem de mare de cereri simultane, deoarece fiecare cerere va recepționa propriul său fir. Atunci când proiectați servicii în Windows NT, programele dumneavoastră vor utiliza în general modelul concurent. Veți utiliza porturile de I/O de completare numai în conjuncție cu aplicațiile care utilizează modelul concurent.

Dar crearea unui serviciu de model concurent este departe de a fi scopul acestei cărți. Este suficient ca dumneavoastră să înțelegeți diferența între modelul serviciilor seriale și cele de model concurent atunci când veți continua dezvoltarea propriilor programe.

Observație: *Puteți utiliza porturile de I/O de completare numai sub Windows NT. Windows 95 nu are funcționalitatea necesară pentru a implementa aceste porturi.*

1496 UTILIZAREA OPERAȚIILOR DE I/O CU ALERTĂ PENTRU PROCESĂRI ASINCRONE



De fiecare dată când o funcție creează un fir de execuție, sistemul creează de asemenea o coadă de mesaje pentru acel fir și o asociază firului. Sistemul de operare mai creează și o altă coadă pentru acel fir, coada APC (*Asynchronous Procedure Call*). Sistemul de operare utilizează funcții de nivel jos din cadrul modulului kernel pentru a crea și manevra coada APC. Din cauza acestor funcții de nivel jos utilizate în întreținerea ei, coada APC este o metodă extrem de rapidă și de eficientă în gestionarea operațiilor de I/O asincrone.

Este posibil ca programele dumneavoastră să efectueze cereri de I/O și funcțiile lor să transmită rezultatele cererilor de I/O direct către coada APC a firului apelant. Pentru a trimite cereri de I/O finalizate către coada APC a firului dumneavoastră, veți utiliza funcțiile *ReadFileEx* și *WriteFileEx*, după cum arătăm mai jos:

```
BOOL ReadFileEx(HANDLE hFile, LPVOID lpBuffer, DWORD
                nNumberOfBytesToRead, LPOVERLAPPED
                lpOverlapped, LPOVERLAPPED_COMPLETION_ROUTINE
                lpCompletionRoutine);
BOOL WriteFileEx(HANDLE hFile, LPVOID lpBuffer, DWORD
                nNumberOfBytesToWrite, LPOVERLAPPED
                lpOverlapped, LPOVERLAPPED_COMPLETION_ROUTINE
                lpCompletionRoutine);
```

După cum puteți vedea, ambele funcții acceptă, ca ultim parametru, adresa unei rutine de încheiere pentru a fi executată atunci când ele își termină acțiunea. Secțiunea 1498 explică funcțiile *ReadFileEx* și *WriteFileEx* în detaliu. Trebuie să utilizați următorul prototip pentru rutina de încheiere pe care ambele funcții o utilizează:

```
VOID WINAPI FileIOCompletionRoutine(
DWORD dwErrorCode,   // codul de incheiere
DWORD dwNumberOfBytesTransfered, //numarul de octeti transferati
LPOVERLAPPED lpOverlapped  // pointer la structura cu
                           // informatii despre I/O
);
```

Parametrul *dwErrorCode* specifică starea de încheiere a operației de I/O. Parametrul *dwErrorCode* poate avea una dintre cele două valori prezentate în tabelul 1496.

Valoare	Semnificație
0	Operația de I/O a fost reușită
ERROR_HANDLE_EOF	Funcția a încercat să citească dincolo de sfârșitul de fișier

Tabelul 1496 *Valorile posibile pentru parametrul* ***dwErrorCode.***

Parametrul *dwNumberOfBytesTransferred* specifică numărul de octeți transferați. Dacă apare vreo eroare, acest parametru este zero. Parametrul *lpOverlapped* indică structura *OVERLAPPED* specificată de funcția de I/O asincronă. Mediul Windows nu utilizează membrul *hEvent* al structurii *OVERLAPPED*; aplicația apelantă poate utiliza acest membru pentru a transmite informații către rutina de încheiere, astfel încât rutina de încheiere să poată dezaloca memoria utilizată de structura *OVERLAPPED*.

Funcția *FileIOCompletionRoutine* ține locul unui nume de funcție definită de aplicație sau de o bibliotecă. Returnearea din funcția *FileIOCompletionRoutine* permite mediului *Windows* apelarea unei alte rutine de încheiere I/O. Toate rutinele de încheiere în așteptare sunt apelate înainte ca așteptarea firului alertabil să fie completată cu un cod de retur *Wait_I/O_COMLETION*. Mediul Windows poate apela în orice ordine rutinele de încheiere din așteptare. El poate sau nu să apeleze rutinele în ordinea în care programul finalizează funcțiile de I/O. De fiecare dată când Windows apelează o rutină de încheiere, el utilizează o parte din stiva aplicației. Dacă rutina de încheiere realizează operații de I/O suplimentare și așteptări cu alertă, stiva poate crește.

OPERAȚIILE DE I/O CU ALERTĂ FUNCȚIONEAZĂ NUMAI ÎN WINDOWS NT

C/C++ 1497

Operațiile de I/O cu alertă sunt o tehnică avansată pentru tratarea operațiilor de I/O asincrone ce utilizează coada de mesaje I/O și una sau mai multe funcții callback. Deoarece operațiile de I/O cu alertă utilizează versiunile extinse ale funcțiilor *ReadFile* și *WriteFile*, programele dumneavoastră pot utiliza aceste operații numai dacă sunteți sigur că ele vor rula pe un sistem Windows NT. Dacă încercați să utilizați funcțiile *ReadFileEx* sau *WriteFileEx* pe un sistem Windows 95 sau Win32, funcțiile vor returna *False* și nu vor efectua nici o procesare. Un apel al funcției *GetLastError* va returna eroarea *ERROR_CALL_NOT_IMPLEMENTED*. Nu încercați să utilizați operații de I/O cu alertă într-un sistem Windows 95 pentru că pot apărea efecte imprevizibile.

UTILIZAREA FUNCȚIILOR READFILEEX ȘI WRITEFILEEX

C/C++ 1498

După cum ați învățat în secțiunea 1494, programele dumneavoastră pot utiliza funcțiile *ReadFileEx* sau *WriteFileEx* pentru a efectua operații de I/O asincrone într-un sistem Windows NT. Funcția *ReadFileEx* citește în mod asincron date dintr-un fișier. Programatorii au proiectat funcția *ReadFileEx* numai pentru operații asincrone, spre deosebire de funcția *ReadFile*, pe care au proiectat-o atât pentru operații sincrone, cât și asincrone. *ReadFileEx* permite unei aplicații să execute alte prelucrări în timpul operației de citire a unui fișier. Funcția *ReadFileEx* își raportează starea de încheiere în mod asincron, apelând o rutină de încheiere specificată de dumneavoastră, atunci când va finaliza procesul de citire și firul de execuție apelant este într-o stare de așteptare cu alertă. În programele dumneavoastră veți utiliza funcția *ReadFileEx* după cum arătăm în următorul prototip:

```
BOOL ReadFileEx(
HANDLE hFile,                    // identificatorul fisierului din
                                 // care citim
LPVOID lpBuffer,                 // adresa bufferului
DWORD nNumberOfBytesToRead,      // numarul de octeti de citit
LPOVERLAPPED lpOverlapped,       // adresa deplasamentului
LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
// adresa rutinei de incheiere
);
```

Funcția *ReadFileEx* acceptă parametrii detaliați în Tabelul 1498.

Parametru	Descriere
hFile	Un identificator de fișier deschis care specifică entitatea fișier din care se va citi. Acest identificator de fișier trebuie creat cu indicatorul *FILE_FLAG_OVERLAPPED* și trebuie să aibă permis accesul de tip *GENERIC_READ*. Parametrul *hFile* poate fi orice identificator pe care funcția *CreateFile* l-a deschis cu indicatorul *FILE_FLAG_OVERLAPPED*.
lpBuffer	Indică un buffer care primește datele citite din fișier. Aplicația nu trebuie să utilizeze acest buffer înainte ca funcția să încheie operația de citire.
nNumberOfBytesToRead	Specifică numărul de octeți care vor fi citiți din fișier.
lpOverlapped	Indică o structură de tipul *OVERLAPPED* care furnizează datele utilizate în timpul operației asincrone (suprapuse) de citire din fișier. Dacă fișierul specificat de *hFile* acceptă deplasamente de octeți, procesul ce apelează funcția *ReadFileEx* trebuie să specifice un deplasament de octeți din cadrul fișierului, de la care să înceapă citirea din fișier. Procesul apelant specifică deplasamentul de octeți prin stabilirea membrilor *Offset* și *OffsetHigh* ai structurii *OVERLAPPED*. Dacă entitatea fișier specificată de *hFile* nu acceptă deplasamente de octeți (de exemplu, dacă este un canal de transfer nominal), procesul apelant trebuie să stabilească membrii *Offset* și *OffsetHigh* la zero, altfel funcția *ReadFileEx* va eșua. Funcția *ReadFileEx* ignoră membrul *hEvent* al structurii *OVERLAPPED*. Funcția *ReadFileEx* își semnalează încheierea operației de citire prin apelarea (sau prin introducerea în coadă a unui apel la) rutinei de încheiere indicată de *lpCompletionRoutine* și, prin urmare, nu necesită un identificator de eveniment. Funcția *ReadFileEx* utilizează membrii Internal și InternalHigh ai structurii OVERLAPPED. Aplicațiile nu trebuie să stabilească acești membri. Structura de date OVERLAPPED indicată de *lpOverlapped* trebuie să rămână validă pe parcursul operației de citire.
lpCompletionRoutine	Indică rutina de încheiere care va fi apelată atunci când operația de citire va fi încheiată și firul de execuție apelant se va afla într-o stare de așteptare cu alertă.

Tabelul 1498 *Parametrii funcției* ***ReadFileEx***.

Dacă funcția *ReadFileEx* reușește, ea va returna o valoare nenulă. Dacă ea eșuează, va returna zero. Pentru a obține informații extinse asupra erorii, apelați funcția *GetLastError*. Dacă funcția se încheie cu succes, firul apelant are o operație de I/O asincronă în așteptare: operația suprapusă de citire din fișier. Atunci când operația de citire suprapusă se încheie și sistemul blochează firul apelant într-o stare de așteptare cu alertă, sistemul apelează funcția spre care indică *lpCompletionRoutine*, iar starea de așteptare se încheie cu codul de retur *WAIT_IO_COMPLETION*.

Dacă funcția reușește și operația de citire din fișier se încheie, dar firul apelant nu se află într-o stare de așteptare cu alertă, sistemul introduce în coada de așteptare apelul rutinei de încheiere, menținând apelul până când firul apelant intră într-o stare de așteptare cu alertă. Dacă *ReadFileEx* încearcă să citească dincolo de sfârșitul unui fișier, funcția va returna zero, iar *GetLastError* va returna *ERROR_HANDLE_EOF*.

Dacă un alt proces blochează o porțiune a fișierului specificat de *hFile*, iar operația de citire specificată în apelul către funcția *ReadFileEx* suprapune porțiunea blocată, apelul funcției *ReadFileEx* eșuează. Dacă funcția *ReadFileEx* încearcă să citească date dintr-un slot pentru mail al cărui buffer este prea mic, funcția returnează *False*, iar *GetLastError* va returna eroarea *ERROR_INSUFFICIENT_BUFFER*. Aplicațiile nu trebuie nici să citească, nici să scrie în bufferul de intrare pe care îl utilizează o operație de citire atâta timp cât operația de citire nu este finalizată. Un acces prematur la bufferul de intrare poate duce la degradarea datelor citite din acel buffer. Funcția *ReadFileEx* mai poate eșua și dacă avem prea multe cereri de operații de I/O asincrone. În astfel de cazuri, funcția GetLastError poate returna *ERROR_INVALID_USER_BUFFER* sau *ERROR_NOT_ENOUGH_MEMORY* (după cum ați învățat în secțiunea 1488).

Dacă încercați să citiți de pe o unitate de disc flexibil care nu conține un disc, sistemul afișează o casetă de mesaj cerându-i utilizatorului să reîncerce operația. Pentru a preveni afișarea de către sistem a acestei casete de mesaj, apelați funcția *SetErrorMode* cu indicatorul *SEM_NOOPENFILEERRORBOX*. Dacă *hFile* este un identificator pentru un canal de transfer nominal sau o altă entitate fișier care nu acceptă conceptul de deplasament de octeți, membrii *Offset* și *OffsetHigh* ai structurii *OVERLAPPED* indicați de parametrul *lpOverlapped* trebuie să fie zero, altfel funcția *ReadFileEx* eșuează.

UTILIZAREA UNEI RUTINE CALLBACK DE ÎNCHEIERE

C/C++1499

După cum ați învățat în secțiunea 1497, programele dumneavoastră trebuie să utilizeze o rutină *callback* de încheiere împreună cu funcțiile *ReadFileEx* și *WriteFileEx*. Trebuie să utilizați următorul prototip pentru rutinele de încheiere apelate de ambele funcții:

```
VOID WINAPI FileIOCompletionRoutine(
    DWORD dwErrorCode, // codul de incheiere
    DWORD dwNumberOfBytesTransfered, // numarul de octeti
                                     // transferat
    LPOVERLAPPED lpOverlapped        // pointer la structura cu
                                     // informatii despre I/O
);
```

Atunci când apelați unul dintre obiectele *Wait* și plasați firul într-o stare cu alertă, sistemul de operare verifică mai întâi coada APC a firului dumneavoastră. Dacă în coadă se află cel puțin o intrare, sistemul nu va pune firul să aștepte; ci în schimb va scoate intrarea din coada APC și firul dumneavoastră apelează rutina callback, transmițând codul de eroare al cererii încheiate de I/O, numărul de octeți transferat și adresa structurii *OVERLAPED* pe care firul a transmis-o inițial cererii de I/O. După ce rutina callback își încheie procesarea, sistemul va verifica din nou dacă mai există intrări în coada APC. Dacă există mai multe intrări, sistemul le va transmite în ordine către rutina callback. Dacă nu mai există intrări, funcția alertă se va returna, iar firul își va continua activitatea fără a aștepta. Prin urmare, singura dată când firul dumneavoastră așteaptă va fi atunci când nu sunt intrări în coada sa APC.

1500 UTILIZAREA UNUI PROGRAM CU INTRĂRI/IEȘIRI CU ALERTĂ



Programele dumneavoastră pot utiliza tehnicile puternice ale operațiilor de I/O cu alertă pentru a efectua prelucrări solide ale intrărilor și ieșirilor asincrone. CD-ROM-ul ce însoțește această carte cuprinde programul *Alertable_IO.cpp*, care utilizează conceptul operațiilor de I/O cu alertă pentru a efectua o sarcină simplă de copiere. Atunci când compilați și executați programul, el va utiliza tehnica de I/O cu alertă pentru a pregăti și a copia fișierul și vă va alerta asupra activităților sale.

Atunci când programul începe, el creează un set de cereri de I/O. Pentru aceasta, el inițializează un set de structuri *MAX_PENDING_IO_REQS*, pe care le va utiliza pentru informarea sistemului de operare asupra numărului maxim de cereri de I/O simultane. Fiecare structură conține o structură *OVERLAPPED*, însă nici una nu conține un pointer la membrul *hEvent*. În plus față de structura *OVERLAPPED* pe care fiecare cerere de I/O o necesită, fiecare cerere necesită și un buffer de memorie, pe care programul îl va menține în cadrul structurii *IO_REQS*.

După ce programul inițializează o structură *IO_REQS*, el apelează funcția *ReadFileEx* pentru a emite cererea de citire a fișierului din sistemul de operare. În acest punct, programul începe să utilizeze operații de I/O cu alertă. Procesul se va returna imediat la promptul casetă de dialog pentru ca utilizatorul să poată introduce imediat un alt fișier pentru a fi copiat. Însă, în fundal, funcția *ReadFileEx* caută și citește fișierul. Atunci când se încheie, funcția alertează procesul (prin intermediul rutinei callback) care utilizează informația pe care a citit-o anterior din fișier pentru a scrie copia fișierului. Cu toate că programul este prea lung pentru a fi prezentat în întregime aici, este util analizați cele două funcții *callback*:

```
void WINAPI WriteCompletionRoutine(DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred, LPOVERLAPPED pOverlapped);
void WINAPI ReadCompletionRoutine(DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred, LPOVERLAPPED pOverlapped)
  {
    PIOREQ pIOReq = (PIOREQ) pOverlapped;
    chASSERT(dwErrorCode == NO_ERROR));
    g_cs.nReadsInProgress--;

    //Rotunjeste numarul de octeti care se vor scrie intr-un sector.
    dwNumberOfBytesTransferred = (dwNumberOfBytesTransferred +
```

```
            g_cs.dwPageSize - 1) & ~(g_cs.dwPageSize - 1);
    chVERIFY(WriteFileEx(g_cs.hFileDst, pIOReq->pbData,
    dwNumberOfBytesTransferred, pOverlapped
            WriteCompletionRoutine));
    g_cs.nWritesInProgress++;
  }
void WINAPI WriteCompletionRoutine(DWORD dwErrorCode, DWORD
dwNumberOfBytesTransferred, LPOVERLAPPED pOverlapped);
  {
    PIOREQ pIOReq = (PIOREQ) pOverlapped;
    chASSERT(dwErrorCode == NO_ERROR));
    g_cs.nWritesInProgress--;
    if (g_cs.ulNextReadOffset.QuadPart < g_cs.ulFileSize.QuadPart)
      {
        // Functia inca nu a citit dincolo de finalul fiserului.
        // Citeste urmatorul segment de date.
        pOverlapped->Offset = g_cs.ulNextReadOffset.LowPart;
        pOverlapped->OffsetHigh = g_cs.ulNextReadOffset.HighPart;
        chVERIFY(ReadFileEx(g_cs.hFileSrc, pIOReq->pbData,
            BUFFSIZE, pOverlapped, ReadCompletionRoutine));
        g_cs.nReadsInProgress++;
        g_cs.ulNextReadOffset.QuadPart += BUFFSIZE;
      break;
    }
}
```

După cum puteţi vedea, funcţia callback *ReadFileCompletionRoutine* apelează funcţia *WriteFileEx* cu informaţia returnată de *ReadFileEx* şi cu adresa funcţiei callback *WriteFileCompletionRoutine.* Funcţia *WriteFileCompletionRoutine* verifică dimensiunea curentă a copiei, de fiecare dată când funcţia *WriteFileEx* returnează valori; când copierea nu este încheiată, *WriteFileCompletionRoutine* schimbă locaţia deplasamentului din cadrul fişierului şi apelează din nou funcţia *ReadFileEx.* Când copierea s-a încheiat, funcţia se termină, iar programul se ocupă în altă parte de operaţiile finale cu fişierele. După cum puteţi vedea, implementarea operaţiilor de I/O cu alertă în programele dumneavoastră este o modalitate relativ simplă şi foarte utilă pentru a efectua operaţii de I/O asincrone.

Observaţie: *Programul* ***Alterable_IO.cpp*** *va rula corect numai într-un sistem Windows NT.*

Index